PHP: PDO – PHP Data Objects – Objetos para acesso simplificado a diferentes bancos de dados

Clube Delphi +PHP

DevMedia group

Ano 9 :: Edição 112 :: R$ 11,90

QuickUpdate
Delphi e XML

Easy Delphi
Curso de POO – Trabalhando com Pacotes, UML e Interfaces

Boas Práticas
Bugs e soluções - Identificação, acompanhamento, correção e prevenção

db Express e Data Snap

Novidades no acesso a dados no Delphi 2010

ISSN 1517990-7

na web ::

Confira no portal ClubeDelphi PLUS 3 vídeo-aulas:

- Geolocalização pelo IP
- Engenharia de Software – Auditoria
- Nova Tool Palette do Delphi 2010 estilo Delphi 7

local
Data Center

Nossos **servidores** dedicados possuem monitoramento 24x7x365, link dedicado e **full duplex**, ou seja, os servidores atingem a **velocidade** do link tanto para **upload** quanto para **download.**

HOST

Servidor
Sites
Gerenciamento

DATACENTER

Co-location
E-mail
Backup

TELECOM

Cross Connection
Banda IP
Link PAP

SOLUÇÕES INTELIGENTES

Solicite um representante comercial

www.localdatacenter.com.br

SOLICITE UM ORÇAMENTO
TEL.: 3956 - 4900
comercial@localdatacenter.com.br

Sumário

Quick Update

5 –Delphi e XML
[*Daniel Sobrinho*]

Delphi

Expert, Boas Práticas

7 – Interfaces
Conheça em detalhes um dos recursos mais poderosos da OO
[*Vitor Luiz Rubio*]

Boa Ideia

20 – DBX e DataSnap
O que há de novo no acesso a dados no Delphi 2010
[*Daniel Laporte*]

Boas Práticas

Engenharia de Software, Boas Práticas

29 – Bugs e soluções
Identificação, acompanhamento, correção e prevenção
[*Ricardo Coelho*]

Easy Delphi

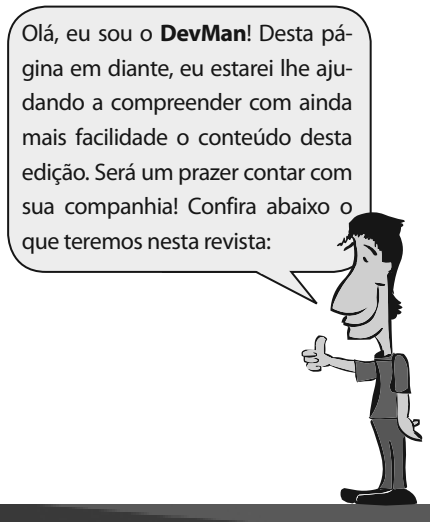
Easy Delphi, Boas Práticas, Tutorial

40 – Introdução à POO – Parte 5
Aprenda de uma forma simples a programar da forma correta
[*Guinther Pauli*]

PHP

PHP

46 – PDO – PHP Data Objects
Interface de acesso a Banco de Dados
[*Rafael Silva*]



Brinde na web desta edição - 3 vídeos! Confira:

3 Vídeos

Geolocalização pelo IP
http://www.devmedia.com.br/articles/viewcomp.asp?comp=14419

Auditoria de sistemas com Together Live Source
http://www.devmedia.com.br/articles/viewcomp.asp?comp=14351

Delphi 2010- Nova Tool Palette estilo Delphi 7
http://www.devmedia.com.br/articles/viewcomp.asp?comp=14339

Para acessar os vídeos utilize os seguintes dados
Login: ***dvm.pl*** *Senha:* ***wwgd21***

Gostou das vídeo aulas? O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Para votar, você vai precisar do código e senha de banca desta edição, que é:
Login: **dvm.pl** Senha: **wwgd21**

Clube Delphi +PHP

Ano 9 - 112ª Edição 2009 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Guinther Pauli
guinther@devmedia.com.br

Editor Técnico
Paulo Quicoli
pauloquicoli@devmedia.com.br

Daniel Sobrinho
daniel.sobrinho@devmedia.com.br

Rodrigo Carreiro Mourão
rodrigocarreiro@rmfactory.com.br

Diagramação e Capa
Romulo Araujo
romulo@devmedia.com.br

Coordenação Geral
Daniella Costa - daniella@devmedia.com.br

Revisão e Supervisão
Thiago Vincenzo - thiago.v.ciancio@devmedia.com.br

Distribuição
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Cristiany Queiroz – Atendimento ao Leitor
www.devmedia.com.br/mancad
(21) 3382-5038

Kaline Dolabella – Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

EDITORIAL

"Cannot Initialize the Borland Database engine". Esta provavelmente foi uma das mensagens de erro mais populares de toda a história do Delphi. Lembro até hoje da primeira aplicação de banco de dados que criei. Ao tentar rodar o EXE em outra máquina, ali estava minha primeira surpresa. Era preciso instalar o BDE na estação! O dbExpress apareceu originalmente no Kylix 1, depois no Delphi 6, e pôs um fim ao reinado do tão popular engine de acesso a banco de dados que perdurou por anos (aproveitando, se você por algum motivo obscuro ainda usa o BDE, confira uma matéria na edição 38). A ideia principal do dbExpress era criar uma "casca" fina em cima dos drivers clientes de acesso a banco de dados, baseando-se em interfaces. Nada de instalar dezenas de bibliotecas. O dbExpress era mais rápido, baseado em cursores unidirecionais, tinha uma íntima relação com o ClientDataSet / DataSetProvider e DataSnap como um todo. Enfim, o acesso a dados no Delphi Win32 se tornou algo realmente profissional, pronto para aplicações comerciais, corporativas e para ambientes multithread, como aplicações multicamadas e Web. Porém, ele tinha um defeito grave: não suportava o Firebird de forma nativa. Aqui estamos, quase no ano de 2010, com o Delphi 2010, e é com alegria que trazemos este mês um artigo sobre o dbExpress no Delphi 2010. Você pode agora, finalmente, usar um driver nativo do DBX para o Firebird. Neste mesmo artigo, confira as novidades do DataSnap 2010: novos wizards, suporte a CallBack, melhor integração com protocolos HTTP, suporte a JSON / REST e muito mais, no artigo do Daniel.

Nesta edição, voltamos novamente a falar em Interfaces na programação orientada a objetos com o Delphi, no artigo do Vitor. Na continuação do mini-curso sobre desenvolvimento OO com Delphi, também comento sobre este brilhante recurso, além de apresentar o suporte a pacotes, UML e como estender o framework criado até aqui.

Ricardo Coelho traz um excelente artigo sobre um tema que é dor de cabeça na certa (e que por coincidência abriu este editorial): BUGs. Conheça as principais exceções mais comuns que acontecem em aplicações Delphi, saiba sua causa e como solucioná-las. Melhor, saiba como gerenciar bugs. Permita ao seu usuário cadastrar bugs e acompanhar sua resolução através de um mecanismo de Bug Tracking totalmente free. E finalmente, faça o Profiling do seu código para detectar gargalos, bugs, memory leaks etc.

Na sessão PHP, o Rafael apresenta a biblioteca PDO – PHP Data Objects, basicamente um framework de acesso a dados para PHP muito semelhante ao que já sabemos fazer com os engines de acesso a dados do Delphi. Finalmente o acesso a dados no PHP está tão robusto como no Delphi!

Sucesso e abraço.

Guinther Pauli
guinther@devmedia.com.br
Certificado Delphi 3,5,6,7,2005,2006,Web,Kylix
MCP,MCAD,MCSD.NET,MCTS,MCPD
guintherpauli.blogspot.com
twitter.com/guintherpauli

ClubeDelphi PLUS

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

DevMedia group

Quick Update: Delphi e XML

Daniel Sobrinho Laporte

daniel.laporte@gmail.com

daniel.sobrinho@devmedia.com.br)

Editor técnico da revista ClubeDelphi. Graduado em Sistemas de informação e pós-graduado em administração de bancos de dados pela Universidade de Franca. É analista de sistemas e desenvolvedor Delphi desde 2005 desenvolvendo aplicações cliente/servidor e multicamadas em Delphi e Firebird.

Como todos nós sabemos, trabalhar com XML no Delphi é muito simples. Se você é um aficionado em ClientDataSets (assim como eu), pode imaginar que facilmente podemos criar uma estrutura XML através deste componente. Porém, neste caso o documento XML gerado possui diferenças de estrutura comparado a um XML comum, isto deve-se ao fato do ClientDataSet armazenar a estrutura em um DataPacket. Devido a este pequeno impasse, muitos desenvolvedores encontram dificuldades ao realizar a leitura de um XML não gerado pelo ClientDataSet, utilizando assim componentes como o XMLTransform por exemplo. Veremos aqui uma maneira simples de realizar a leitura de um arquivo estruturado, utilizando para isto o componente XMLDocument. O layout do formulário pode ser visto na **Figura 1**. Adotaremos um caso de estrutura pré-definida para a reprodução, conforme ilustra o exemplo da **Listagem 1**.

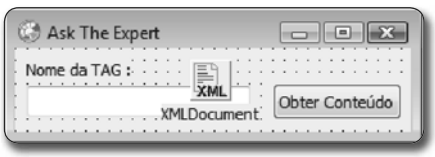


Figura 1. Layout do Exemplo

Observando a **Listagem 2**, temos entre as Linhas 03 e 05 a declaração das variáveis necessárias. Repare que na Linha 05 definimos três tipos IXMLNode que nada mais é do que um nó

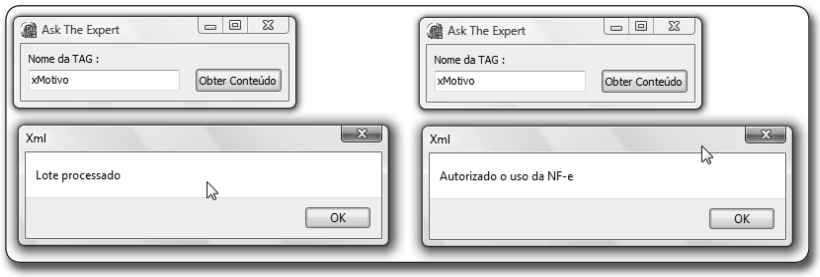


Figura 2. Resultado da aplicação em funcionamento

Listagem 1. Documento XML para leitura

```
<?xml version="1.0" encoding="UTF-8" ?>
<retConsReciNFe xmlns="http://www.portalfiscal.inf.br/nfe"
versao="1.10">
  <tpAmb>2</tpAmb>
  <verAplic>SP_NFE_PL_005d</verAplic>
  <nRec>1</nRec>
  <cStat>104</cStat>
  <xMotivo>Lote processado</xMotivo>
  <cUF>35</cUF>
 <protNFe versao="1.10">
  <infProt>
  <tpAmb>2</tpAmb>
  <verAplic>SP_NFE_PL_005d</verAplic>
  <chNFe>123456789</chNFe>
  <dhRecbto>2009-09-15T17:46:42</dhRecbto>
  <nProt>9876543210</nProt>
  <digVal>ABCDEFGHIJKLMNOPQRSTUVXZdigVal>
  <cStat>100</cStat>
  <xMotivo>Autorizado o uso da NF-e</xMotivo>
  </infProt>
 </protNFe>
</retConsReciNFe>
```

de XML. As linhas 07 a 13 definem o teste de preenchimento do texto no Edit fazendo com que a rotina seja finalizada caso este esteja em branco (Linha 12). Perceba que na Linha 14 definimos um contador que será incrementado de acordo com o número de ocorrências. Na Linha 15 referenciamos a variável stPath com o caminho do XML (considerando que se encontra no mesmo diretório da aplicação), realizando em seguida o teste de existência do arquivo. Nas Linhas 19 a 21 carregamos o arquivo XML. Como o XML é um documento estruturado e baseado em Tag's, o nosso objetivo é percorrer todas estas Tag's, exatamente como fazemos na Linha 22. Note que a cada vez que é realizada a leitura de uma Tag a variável Node recebe o nó em questão e compara com o texto digitado no Edit (Linhas 24 e 25). Caso não seja encontrado, é realizada uma nova contagem na qual analisa se aquela Tag possui Childs. Mais uma vez, a Tag é carregada (desta vez na variável NodeChild como mostra a Linha 37). Utilizamos o método FindNode (Linha 38) para verificar a existência da Tag filha, obtendo assim um ponteiro de referência à estrutura encontrada. Em ambas as situações, ao encontrar o Child o contador é incrementado e o conteúdo é exibido em forma de mensagem. Caso contrário, teremos a mensagem de que não foi encontrada a Tag, pois a variável x terá seu valor igual a zero (Linhas 50 e 51). Execute a aplicação, testando a Tag xMotivo e veja que serão exibidas duas mensagens, pois a Tag está presente duas vezes no documento.

Executando a aplicação, digitamos o nome de um elemento (tag) no Edit e obtemos todos os valores contidos encontrados. O resultado é mostrado na **Figura 2.** ●

Listagem 2. Evento OnClick do btnXMLClick

```
01 procedure TfrmPrincipal.btnXMLClick(Sender: TObject);
02 var
03   stPath : String;
04   i, j, x : Integer;
05   Node, NodeChild, NodeAux : IXMLNode;
06 begin
07   if Trim(edtTag.Text) = '' then
08   begin
09     ShowMessage('Digite o nome da TAG!');
10     if edtTag.CanFocus then
11       edtTag.SetFocus;
12     Exit;
13   end;
14   x := 0;
15   stPath := ExtractFilePath(Application.ExeName) + 'Arquivo.xml';
16   if FileExists(stPath) then
17   begin
18     try
19       XMLDocument.Active   := False;
20       XMLDocument.FileName := stPath;
21       XMLDocument.Active   :=  True;
22       for i := 0 to XMLDocument.DocumentElement.ChildNodes.
          Count - 1 do
23       begin
24         Node := XMLDocument.DocumentElement.ChildNodes[i];
25         if Node.NodeName = Trim(edtTag.Text) then
26         begin
27           if not VarIsNull(Node.NodeValue) then
28           begin
29             ShowMessage(VarToStr(Node.NodeValue));
30             Inc(x);
31           end;
32         end
33         else
34         begin
35           for j := 0 to Node.ChildNodes.Count - 1 do
36           begin
37             NodeChild := Node.ChildNodes[j];
38             NodeAux    := NodeChild.ChildNodes.
                FindNode(edtTag.Text);
39             if Assigned(NodeAux) then
40             begin
41               if not VarIsNull(NodeAux.NodeValue) then
42               begin
43                 ShowMessage(VarToStr(NodeAux.NodeValue));
44                 Inc(x);
45               end;
46             end;
47           end;
48         end;
49       end;
50       if x = 0 then
51        ShowMessage('Tag não encontrada!');
52     except
53       on e: exception do
54         ShowMessage('Problemas ao ler a tag!');
55     end;
56   end;
57 end;
```

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

Interfaces

Conheça em detalhes um dos recursos mais poderosos da OO

Do que trata o artigo
Abordaremos o básico sobre Interfaces, a sua importância na POO, maneira correta de usar e como construir classes que podem ser usadas com interfaces. Trataremos também os benefícios de se usar classes já prontas do Delphi para se trabalhar com interfaces e como fazer com que objetos deleguem a implementação de uma interface a um outro objeto contido ou associado a ele.

Para que serve
Sendo uma técnica de programação orientada a objetos, as funcionalidades abordadas no artigo terão finalidade mais didática do que prática. Uma exceção é o uso de TAggregatedObject e TContainedObject, que pode demonstrar como objetos que implementam uma interface podem delegar essa tarefa a objetos agregados que realmente a implementam, economizando muitas linhas de código na confecção de wrappers.

Em que situação o tema é útil
Sempre que criarmos interfaces similares, porém sem grau de parentesco entre si, mas mesmo assim relacionadas. Por exemplo, quando desejamos encapsular a implementação de uma interface A em uma classe B que já implementa a interface B, sem precisar implementar os métodos da Interface A explicitamente, muito menos precisar tornar esta classe B um wrapper de outra classe A qualquer que implemente a interface A, economizando tempo e evitando códigos que misturem conceitos e responsabilidades.

Resumo do DevMan

Interfaces são essenciais nos nossos dias. Quem programa orientado a interfaces consegue criar sistemas altamente escalonáveis, com baixíssimo grau de acoplamento e de fácil manutenção. As interfaces ainda favorecem a separação de conceitos e responsabilidades e agilizam a programação em equipe, tornando as partes do sistema como "peças de Lego" que podem ser encaixadas de diversas formas.

O uso de interfaces também favorece um ambiente pseudo – gerenciado, onde o desenvolvedor precisa se preocupar em instanciar objetos, mas os contadores automáticos de referências do Delphi garantem que os mesmos serão destruídos da memória quando não forem mais necessários.

Vitor Luiz Rubio
vitorrubio@gmail.com
Analista de Sistemas Sr. na Editora Revista dos Tribunais. Trabalha com Delphi desde a versão 3. Formado em Processamento de Dados pela FATEC-SP.

O paradigma da orientação a objetos nos ajuda a criar programas mais robustos, flexíveis e escalonáveis, isto é um fato. Mas um recurso que todas as linguagens orientadas a objeto possuem e poucas pessoas utilizam são as interfaces.

Interfaces ajudam a separar o desenvolvimento das classes ou módulos entre vários programadores, inclusive de empresas diferentes. Basta que as interfaces sejam combinadas ou convencionadas corretamente, como um contrato, e que tenham métodos com escopo, objetivos, entradas e saídas bem definidas.

Isso permite padronizar as interações e trocas de mensagens entre nossos objetos, e permite também criar objetos de linhagens diferentes que obedeçam as mesmas interfaces, para que possam ser intercambiáveis.

Num contexto dinâmico onde as regras de negócio podem ser modificadas ou o produto ser totalmente remodelado antes mesmo de ser posto em produção, o uso de interfaces nos garante uma grande flexibilidade. Separando nossos conceitos em partes encaixáveis nós podemos mapear facilmente quais implementações modificar, para atender uma regra de negócio sem comprometer um contrato, e quais interfaces devem ser estendidas através de herança.

Interfaces são um dos mais avançados recursos da Programação Orientada a Objetos e permitem mais um nível de abstração na sua análise. O Delphi suporta interfaces desde a versão 3, que na época foram desenvolvidas principalmente para integração com o COM.

Além dos objetivos mencionados, as interfaces também servem para simular herança múltipla. Você já deve ter ouvido falar que a linguagem C++, por exemplo, implementa a herança múltipla real, permitindo que uma classe possa ter duas ou mais classes base. Já o Delphi e o Java não possuem o recurso da herança múltipla para classes, mas possuem o recurso das interfaces, então embora uma classe só possa ser filha de outra classe, uma interface pode ser filha de duas ou mais interfaces ao mesmo tempo. Sendo assim uma classe que implementa uma interface filha de outras três interfaces pode ser considerada como descendente de múltiplas classes. A única forma de simular herança múltipla no Delphi seria através de interfaces.

Uma interface pode ser implícita, por exemplo, todo o conjunto de métodos e propriedades públicas de uma classe são a única interface que ela implementa implicitamente, mesmo que ela não implemente nenhuma interface explicitamente.

Todos os descendentes de uma mesma classe abstrata implementam a mesma interface implicitamente. Tanto é que na época do Delphi 2 e anteriores era possível usar COM sem o recurso das interfaces, usando classes abstratas no lugar. Lógico que a herança múltipla caía por terra.

Dito isto, você também pode entender uma interface como uma classe abstrata, que só tem a definição e não tem implementação, porém tem todos os recursos da POO, como herança e polimorfismo, além do recurso de contagem de referências e do identificador único guid.

Interfaces podem ser usadas para obrigar ou forçar que uma determinada classe tenha, ou implemente, determinados métodos. É isso que torna objetos de linhagens diferentes compatíveis entre si.

Objetos que implementam interfaces, caso sejam descendentes de TInterfacedObject não necessitam ser destruídos, pois tem um contador de referências interno que automaticamente destrói o objeto que a implementa quando esse contador chega a zero. Além disso, esta classe TInterfacedObject tem um mecanismo para trabalhar com multithread que previne que um objeto seja destruído quando ainda não terminou de ser criado.

Então, teoricamente usando interfaces e criando nossas classes como descendentes de TInterfacedObject não precisamos nos preocupar com a destruição dos objetos e nem com Memory Leaks.

Trabalhando com interfaces

Vamos aprender alguns tópicos avançados sobre interfaces, como delegar a implementação de uma interface, como usar Aggregates e evitar memory leaks. Para fazer os nossos testes crie uma VCL Forms Application normal, com um único formulário chamado frmPrincipal e sua unit com o nome uFrmPrincipal.pas.

Todas as interfaces e classes que criaremos nós o faremos nessa unit, por simplicidade, e todos os testes que realizarmos serão executados em botões no formulário principal. Para podermos usar a função ShowMessage, assegure-se que a unit dialogs esteja citada na seção uses da sua unit uFrmPrincipal. Abra o arquivo de projeto, através do menu Project>View source e a seguinte linha de código que ativa um recurso muito interessante, existente desde a versão 2006, que é a detecção desses memory leaks:

```
ReportMemoryLeaksOnShutdown := true;
```

Essa variável de sistema faz com que o Delphi, ao encerrar a execução de um aplicativo, verifique a existência de algum vazamento de memória e reporta ao usuário o ocorrido. Dessa forma, o bloco de código (begin...end) deve se parecer como a seguir:

```
begin
  ReportMemoryLeaksOnShutdown := true;
  Application.Initialize;
  Application.CreateForm(TfrmPrincipal, frmPrincipal);
  Application.Run;
end.
```

Agora na unit uFrmPrincipal adicione a interface e classes exibidas na **Listagem1**.

Na **Listagem 1** criamos uma interface bem simples que define um método chamado Mensagem. Essa interface é implementada por duas classes que herdam de TInterfacedObject para terem os benefícios anteriormente citados. Contudo, o método Mensagem é implementado de forma diferente uma da outra.

Nota

Para mais informações sobre como trabalhar com interfaces, sugiro a leitura das edições 74 e 75 da revista Clube Delphi.

Nota

Um memory leak, ou vazamento de memória, é o termo dado para um objeto que foi criado em tempo de execução, contudo, após seu devido uso e descarte, não é destruído – eliminado da memória.

Nota

Não esqueça do GUID. Ao criar uma interface, pressione ctrl+shift+G para gerar uma GUID, que é uma sequência numérica como esta: ['{37F2C629-0DF6-43FF-B1FC-4A7419DB893E}'].
Isto é necessário para que as nossas interfaces tenham uma identidade única.

Agora vamos ver como utilizar essas classes no lugar da Interface. Coloque no formulário um botão para instanciarmos e testarmos nossos objetos, conforme a **Listagem 2.**

Listagem 1. Básico de Interfaces

```
ITeste = interface
  ['{37F2C629-0DF6-43FF-B1FC-4A7419DB893E}']
  function Mensagem: string;
end;

TTeste = class(TInterfacedObject, ITeste)
 public
  function Mensagem: string;
 end;

TProva = class(TInterfacedObject, ITeste)
 public
  function Mensagem: string;
end;

{ TTeste }

function TTeste.Mensagem: string;
begin
  Result := 'Eu implemento Iteste';
end;

{ TProva }

function TProva.Mensagem: string;
begin
    Result := 'Eu também implemento Iteste, mesmo não sendo
parente de TTeste';
end;
```

Listagem 2. Botão btTeste

```
procedure TForm1.btTesteClick(Sender: TObject);
var
  i1, i2: ITeste;
begin

  i1 := TTeste.Create;
  ShowMessage(i1.Mensagem);

  i2 := TProva.Create;
  ShowMessage(i2.Mensagem);

  //repare que eu não preciso destruir nenhum dos dois objetos.

end;
```

Observe na **Listagem 2** que temos duas variáveis, i1 e i2 que são do tipo ITeste, nossa interface. Na sequência estamos instanciando um objeto que implementa ITeste, um para cada variável. Execute a aplicação e veja que, embora as duas variáveis sejam do mesmo tipo, a implementação é diferente, ou seja, eles fazem a mesma coisa, que é implementar o método Mensagem, porém o fazem de forma diferente.

Vamos agora verificar o momento em que esses objetos são destruídos. Toda classe herda de TObject, que por sua vez possui o método Destroy que pode ser utilizado para se realizar alguma operação quando o objeto é destruído da memória. No nosso caso vamos sobrecarregar método Destroy das suas classes para mostrar o momento exato em que elas são destruídas automaticamente. Isso é feito na **Listagem 3**.

Listagem 3. Ver o momento da destruição de interfaces

```
TTeste = class(TInterfacedObject, ITeste)
 public
   function Mensagem: string;
   destructor Destroy; virtual; override;
end;

TProva = class(TInterfacedObject, ITeste)
 public
   function Mensagem: string;
   destructor Destroy; virtual; override;
end;


destructor TTeste.Destroy;
begin
  ShowMessage('TTeste Destruído!');
  inherited;
end;

function TTeste.Mensagem: string;
begin
  Result := 'Eu implemento Iteste';
end;

{ TProva }

destructor TProva.Destroy;
begin
  ShowMessage('TProva Destruído!');
  inherited;
end;

function TProva.Mensagem: string;
begin
  Result := 'Eu também implemento Iteste, mesmo não sendo
parente de TTeste';
end;
```

Ao executar você vai perceber que logo que chega ao fim do escopo do evento OnClick do botão, os dois objetos são destruídos. Tente agora criar i1 e i2 como variáveis globais. Para isso tire eles da seção var do evento OnClick e coloque na seção var global, logo abaixo de frmPrincipal: TfrmPrincipal, conforme podemos ver a seguir:

```
var
  frmPrincipal: TfrmPrincipal;
  i1, i2: Iteste;
```

Se você clicar no botão uma única vez e depois fechar a janela, verá que, mesmo as mensagens de destruição dos objetos não sendo exibidas, os objetos são destruídos. Podemos saber isso porque nenhum memory leak foi acusado. Se você clicar no botão várias vezes, cada novo objeto criado vai sobrescrever a referência antiga de i1 e i2. Como a referência antiga será perdida, o contador de referências de cada objeto, que está em 1 atingirá 0 antes que o novo objeto seja criado, fazendo com que o objeto antigo seja destruído logo antes do objeto novo ser criado. Acompanhe o passo a passo com o debug do Delphi para entender melhor.

Agora no nosso teste vamos adicionar mais duas linhas, para apagar explicitamente a referência aos objetos, depois de usados, conforme a **Listagem 4**.

Nota

Todas essas classes para se trabalhar com interfaces estão na unit System.pas do Delphi, que possui seu código fonte inteiro, e comentado, disponível. Os comentários explicam bem porque certas coisas foram implementadas daquela forma. E você encontrará bastante código em assembly em alguns métodos.

Delegações e a palavra reservada implements

Há uma maneira de dizermos que uma classe implementa uma determinada interface, mas passar a responsabilidade dessa implementação a um objeto qualquer que será conectado ou agregado à classe, isso se chama delegação. O objeto que será conectado à classe pode implementar a interface ou delegar essa tarefa a outro, e assim por diante. Para entendermos como isso funciona, vamos criar um contexto teórico, que posteriormente será representado por código.

Para jogar games com uma alta demanda gráfica, desses novos que abusam do seu hardware e dos recursos do DirectX, você precisa de uma boa placa de vídeo PCI Express off board. Naturalmente o seu computador, considerando o conjunto placa-mãe, processador, memória e sistema operacional, implementa todas as interfaces necessárias para se jogar um bom game, inclusive a parte de renderização de gráficos. Porém essa parte de renderização de gráficos ele delega para a placa de vídeo PCI Express.

Na sua placa mãe existe um slot PCI Express que segue um padrão bem conhecido de mercado, que é uma convenção. E nesse slot você pode encaixar qualquer placa que siga esse padrão, ou seja, implemente essa interface. A qualidade e eficiência da renderização dos gráficos vai depender não somente da marca, mas do modelo da placa. Sabemos que existem placas para todos os tipos de usuários e bolsos, desde aqueles que não jogam nenhum jogo, passando por aqueles que jogam apenas simples games casuais e terminando com aqueles jogadores mais fiéis, que gostam de jogar os últimos lançamentos com velocidade total, máxima resolução e todos os recursos gráficos no máximo.

Para ilustrar, vamos criar uma classe TComputador que delega a renderização de gráficos para uma propriedade do tipo IPlacaGrafica. Obviamente um modelo mais detalhado para simular um computador rodando jogos deveria considerar cada peça, cada slot e cada interface de um computador verdadeiro, até os parafusos, mas não é esse o nosso objetivo. Nosso objetivo é ilustrar as interações entre o computador como um todo incluindo o sistema operacional e a placa gráfica da maneira mais simples possível. O computador simplesmente dirá à placa: "Renderize isso".

No nosso sistema hipotético tanto as classes de computadores como as classes de placas gráficas terão internamente um campo privado de instância que terá seu valor alterado no construtor da classe. Esse campo é uma string que conterá a mensagem exibida ao usuário.

Vamos adicionar na unit do formulário principal as interfaces e classes da **Listagem 5** e implementação dos métodos conforme a **Listagem 6**.

Na **Listagem 5** definimos as interfaces que representam uma placa gráfica e um computador. Em seguida criamos três classes que implementam IPlacaGrafica, cada uma a sua maneira. Essa implementação pode ser vista na **Listagem 6**. Olhando novamente para **Listagem 5** vemos que a classe TComputador além

Listagem 4. Botão Teste

```
procedure TfrmPrincipal.btTesteClick(Sender: TObject);
begin

  i1 := TTeste.Create;
  ShowMessage(i1.Mensagem);

  i2 := TProva.Create;
  ShowMessage(i2.Mensagem);

  //apaga explicitamente a referência aos objetos atribuindo-
  lhes nil
  i1 := nil;
  i2 := nil;

end;
```

Listagem 5. Interfaces e classes

```
  IPlacaGrafica = interface
    ['{AC3D7441-80F3-4FEC-BA17-E74333BF5FB3}']
    procedure Renderizar;
  end;

  IComputador = interface
    ['{B8D3F074-32D2-4555-B3F3-AC60EB93064C}']
    procedure Inicializar;
  end;

  TPlacaGraficaATI = class(TInterfacedObject, IPlacaGrafica)
  private
    FMensagem: string;
  public
    procedure renderizar;
    constructor Create;
  end;

  TPlacaGraficaNVidia = class(TInterfacedObject, IPlacaGrafica)
  private
    FMensagem: string;
  public
    procedure renderizar;
    constructor Create;
  end;

  TPlacaGraficaProprietaria = class(TInterfacedObject,
   IPlacaGrafica)
  private
    FMensagem: string;
  public
    procedure renderizar;
    constructor Create;
  end;

  TComputador = class (TInterfacedObject, IComputador,
   IPlacaGrafica)
  private
    FPlacaGrafica: IPlacaGrafica;
    FMensagem: string;
  public
    procedure inicializar;
    constructor Create;
    property PlacaGrafica: IPlacaGrafica read FPlacaGrafica
     write FPlacaGrafica implements IPlacaGrafica;
  end;

  TVideogame = class (TInterfacedObject, IComputador,
   IPlacaGrafica)
  private
    FPlacaGrafica: IPlacaGrafica;
    FMensagem: string;
  public
    procedure inicializar;
    constructor Create;
    property PlacaGrafica: IPlacaGrafica read FPlacaGrafica
     write FPlacaGrafica implements IPlacaGrafica;
  end;
```

Listagem 6. Implementação

```
{ TPlacaGraficaATI }

constructor TPlacaGraficaATI.Create;
begin
  inherited;
  FMensagem := 'Os gráficos estão sendo renderizados pela placa
   ATI!'+#13#10;
end;

procedure TPlacaGraficaATI.renderizar;
begin
  ShowMessage(FMensagem);
end;

{ TPlacaGraficaNVidia }

constructor TPlacaGraficaNVidia.Create;
begin
  inherited;
  FMensagem := 'Os gráficos estão sendo renderizados pela placa
   NVIDIA!'+#13#10;
end;

procedure TPlacaGraficaNVidia.renderizar;
begin
  ShowMessage(FMensagem);
end;

{ TPlacaGraficaProprietaria }

constructor TPlacaGraficaProprietaria.Create;
begin
  inherited;
  FMensagem := 'Os gráficos estão sendo renderizados por uma placa
   Proprietaria!'+#13#10;
end;

procedure TPlacaGraficaProprietaria.renderizar;
begin
  ShowMessage(FMensagem);
end;
```

```
{ TComputador }

constructor TComputador.Create;
begin
  inherited;
  FMensagem := 'Inicializando Computador!'+#13#10;
end;

procedure TComputador.inicializar;
begin

  if FPlacaGrafica <> nil then
  begin
    showmessage(FMensagem);
    FPlacaGrafica.renderizar;
  end
  else
    showmessage(FMensagem + 'Impossível renderizar gráficos sem
     uma placa gráfica!' + #13#10);
end;

{ TVideogame }

constructor TVideogame.Create;
begin
  inherited;
  FMensagem := 'Inicializando Videogame!'+#13#10;
end;

procedure TVideogame.inicializar;
begin

  if FPlacaGrafica <> nil then
  begin
    showmessage(FMensagem);
    FPlacaGrafica.renderizar;
  end
  else
    showmessage(FMensagem + 'Impossível renderizar gráficos sem
     uma placa gráfica!' + #13#10);
end;
```

de implementar a interface IComputador, também implementa IPlacaGrafica. Contudo não encontramos nela implementação alguma do método Renderizar. Isso é possível porque TComputador está delegando a algum objeto essa tarefa. Esse objeto está definido na propriedade PlacaGrafica, que em sua declaração possui a palavra chave implements seguida da interface IPlacaGrafica. Isso quer dizer que, embora TComputador implemente IPlacaGrafica, essa implementação não será feita por ele, mas pelo objeto instanciado e armazenado na propriedade PlacaGrafica. O mesmo acontece para a classe TVideoGame. Vamos colocar agora tudo isso para funcionar.

Crie mais um botão no formulário, chame-o de btPlacaGrafica e defina o seu Caption para "Gráficos". Coloque o código da **Listagem 7** em seu evento OnClick.

Como pode ser visto na **Listagem 7** estamos simulando um funcionamento de computador. Instanciamos cada tipo de placa, representadas pelos objetos radeon, geforce e proprio. Também instanciamos as classes TComputador e TVideoGame. Em seguida executamos o método Inicializar de cada computador e do vídeo game para mostrar que é obrigatório ligar uma placa de vídeo a um computador, ou seja, é necessário passar a referência de uma interface IPlacaGrafica para o computador. Na sequência fazemos isso e aí sim, a inicialização ocorre com sucesso. Com isso, chamamos o método Renderizar, primeiramente das placas gráficas e posteriormente dos próprios objetos TComputador e TVideogame, para mostrar que a delegação está funcionando. Por fim, é necessário limpar da memória os objetos computador1, computador2, computador3 e videogame. Isso porque não estão armazenados em interfaces mas em objetos comuns.

Até agora estamos usando objetos descendentes de TInterfacedObject e usamos delegações sem criar nenhum memory leak. Podemos mudar este código para usar somente interfaces e se aproveitar do mecanismo de contagem de referência. Veja na **Listagem 8** que retiramos a parte onde chamamos os métodos Free dos objetos e passamos a referenciar os computadores em interfaces.

Cuidado ao misturar referências de objetos entre interfaces e variáveis. Embora no nosso caso fizemos tudo direitinho, se por um acaso você se descuidar e liberar da memória de um objeto que vai ser liberado automaticamente depois, ou até usado, teremos problemas. Pior ainda quando um objeto é liberado automaticamente porque nenhuma variável do tipo interface aponta mais para ele (contador de referência chega a 0) mas você ainda tem uma variável do tipo objeto apontando para o mesmo. Se você usar essa variável terá problemas de access violation.

Podemos eliminar essa mistura de variável com interface colocando na nossa interface IComputador a propriedade PlacaGrafica e os métodos acessores, mudando nossas interfaces como na **Listagem 9**, e implementando nosso código como na **Listagem 10**.

Observe na **Listagem 9** que agora a interface IComputador expõe uma propriedade que referencia IPlacaGrafica, o que

Listagem 7. Evento OnClick do botão "Gráficos"

```
var computador1, computador2, computador3: TComputador;
  videogame: TVideogame;
  radeon,  gforce, proprio: IPlacaGrafica;
begin

  computador1 := TComputador.Create;
  computador2 := TComputador.Create;
  computador3 := TComputador.Create;
  videogame := TVideogame.Create;

  radeon := TPlacaGraficaATI.Create;
  gforce := TPlacaGraficaNvidia.Create;
  proprio := TPlacaGraficaProprietaria.Create;

  //antes de acoplar a placa gráfica não podemos renderizar
  showmessage('=== inicializando sem placa ===');
  computador1.inicializar;
  computador2.inicializar;
  computador3.inicializar;
  videogame.inicializar;

  //acoplando as placas gráficas
  showmessage('=== acoplando as placas ===');
  computador1.PlacaGrafica :=  radeon;
  computador2.PlacaGrafica :=  gforce;
  computador3.PlacaGrafica := gforce; //esta gforce terá o
   contador de referências = 2, porque dois computadores usam a
   mesma placa (impossível na vida real)
  videogame.PlacaGrafica :=  proprio;

  //depois das placas devidamente acopladas
  showmessage('=== inicializando COM as placas ===');
  computador1.inicializar;
  computador2.inicializar;
  computador3.inicializar;
  videogame.inicializar;

  //renderizando pela propriedade PlacaGrafica
  showmessage('=== Renderizando pela propriedade PlacaGrafica ===');
  computador1.PlacaGrafica.Renderizar;
  computador2.PlacaGrafica.Renderizar;
  computador3.PlacaGrafica.Renderizar;
  videogame.PlacaGrafica.Renderizar;

  //será que o Tcomputador e o TVideogame implementam
   IPlacaGrafica? Poderemos renderizar gráficos novamente, depois
    de inicializado?
  showmessage('=== Renderizando com Typecast ===');
  (computador1 as IPlacaGrafica).Renderizar;
  (computador2 as IPlacaGrafica).Renderizar;
  (computador3 as IPlacaGrafica).Renderizar;
  (videogame as IPlacaGrafica).Renderizar;

  //como nossos computadores são objetos e não interfaces, temos
   de destruí-los, mas as placas não
  computador1.Free;
  computador2.Free;
  computador3.Free;
  videogame.Free;
end;
```

Listagem 8. Usando somente Interfaces

```
procedure TfrmPrincipal.btPlacaGraficaClick(Sender: TObject);
var computador1, computador2, computador3: TComputador;
  videogame: TVideogame;
  radeon,  gforce, proprio: IPlacaGrafica;
  iComp1,iComp2,iComp3,iComp4: IComputador;
begin
  computador1 := TComputador.Create;
  computador2 := TComputador.Create;
  computador3 := TComputador.Create;
  videogame := TVideogame.Create;

  iComp1 := computador1;
  iComp2 := computador2;
  iComp3 := computador3;
  iComp4 := videogame;

  radeon := TPlacaGraficaATI.Create;
  gforce := TPlacaGraficaNvidia.Create;
  proprio := TPlacaGraficaProprietaria.Create;

  //antes de acoplar a placa gráfica não podemos renderizar
  showmessage('=== inicializando sem placa ===');
  iComp1.inicializar;
  iComp2.inicializar;
  iComp3.inicializar;
  iComp4.inicializar;

  //acoplando as placas gráficas
  showmessage('=== acoplando as placas ===');
  computador1.PlacaGrafica :=  radeon;
  computador2.PlacaGrafica :=  gforce;
  computador3.PlacaGrafica := gforce; //esta gforce terá o
   contador de referências = 2, porque dois computadores usam a
   mesma placa (impossível na vida real)
  videogame.PlacaGrafica :=  proprio;

  //depois das placas devidamente acopladas
  showmessage('=== inicializando COM as placas ===');
  iComp1.inicializar;
  iComp2.inicializar;
  iComp3.inicializar;
  iComp4.inicializar;

  //renderizando pela propriedade PlacaGrafica
  showmessage('=== Renderizando pela propriedade PlacaGrafica ===');
  computador1.PlacaGrafica.Renderizar;
  computador2.PlacaGrafica.Renderizar;
  computador3.PlacaGrafica.Renderizar;
  videogame.PlacaGrafica.Renderizar;

  //será que o Tcomputador e o TVideogame implementam
  IPlacaGrafica? Poderemos renderizar gráficos novamente, depois
   de inicializado?
  showmessage('=== Renderizando com Typecast ===');
  (iComp1 as IPlacaGrafica).Renderizar;
  (iComp2 as IPlacaGrafica).Renderizar;
  (iComp3 as IPlacaGrafica).Renderizar;
  (iComp4 as IPlacaGrafica).Renderizar;
end;
```

nos obriga a implementá-la nas classes TComputador e TVideogame, como mostra **Listagem 10.** Com essa abordagem você pode criar um código que use apenas interfaces, evitando riscos e prevenindo leaks. Não precisando se preocupar com a desalocação, conforme a **Listagem 11**.

Veja que agora não é necessário liberar da memória os objetos criados, isso se dará de forma automática. Parece estranho que os computadores 2 e 3 usem a mesma placa de vídeo, pois isso é impossível na vida real, mas fizemos isso propositalmente para ilustrar como uma mesma instância de um objeto pode ser compartilhada por duas instâncias de um outro objeto que delegam alguma tarefa a ela. É aí que se encontra o maior perigo de se misturar variáveis do tipo objeto e interfaces. Para exemplificar esse perigo, vamos criar mais duas interfaces e duas classes conforme o código da **Listagem 12**.

Repare que essas interfaces e classes são iguais as que criamos anteriormente para computador e placa de vídeo, apenas fizemos um ajuste para mudar os nomes dos métodos, classes e propriedades, e também mudamos as mensagens. Veja também que a interface IObjetoQueUsaRecurso expõe uma propriedade do tipo IRecurso compartilhado. Ou seja, estamos trabalhando somente com interfaces. Vamos testar essa utilização.
Adicione mais três botões chamados btRecComp, btCompVar e btComInt com os Captions "Recurso Compartilhado", "Compartilhado com variável" e "Compartilhado com Interface", respectivamente, conforme a **Figura 1**.

Listagem 9. Modificação nas interfaces e classes

```
IPlacaGrafica = interface
  ['{AC3D7441-80F3-4FEC-BA17-E74333BF5FB3}']
  procedure Renderizar;
end;

IComputador = interface
  ['{B8D3F074-32D2-4555-B3F3-AC60EB93064C}']
  procedure Inicializar;
  function GetPlacaGrafica: IPlacaGrafica;
  procedure SetPlacaGrafica(const value:IPlacaGrafica);
  property PlacaGrafica: IPlacaGrafica read GetPlacaGrafica
   write SetPlacaGrafica;
end;

TPlacaGraficaATI = class(TInterfacedObject, IPlacaGrafica)
private
  FMensagem: string;
public
  procedure renderizar;
  constructor Create;
end;

TPlacaGraficaNVidia = class(TInterfacedObject, IPlacaGrafica)
private
  FMensagem: string;
public
  procedure renderizar;
  constructor Create;
end;

TPlacaGraficaProprietaria = class(TInterfacedObject,
 IPlacaGrafica)
private
  FMensagem: string;
public
  procedure renderizar;
  constructor Create;
end;

TComputador = class (TInterfacedObject, IComputador,
 IPlacaGrafica)
private
  FPlacaGrafica: IPlacaGrafica;
  FMensagem: string;
public
  procedure inicializar;
  constructor Create;
  function GetPlacaGrafica: IPlacaGrafica;
  procedure SetPlacaGrafica(const value:IPlacaGrafica);
  property PlacaGrafica: IPlacaGrafica read GetPlacaGrafica
   write SetPlacaGrafica  implements IPlacaGrafica;
end;

TVideogame = class (TInterfacedObject, IComputador,
 IPlacaGrafica)
private
  FPlacaGrafica: IPlacaGrafica;
  FMensagem: string;
public
  procedure inicializar;
  constructor Create;
  function GetPlacaGrafica: IPlacaGrafica;
  procedure SetPlacaGrafica(const value:IPlacaGrafica);
  property PlacaGrafica: IPlacaGrafica read GetPlacaGrafica
   write SetPlacaGrafica  implements IPlacaGrafica;
end;
```

Listagem 10. Implementação das novas interfaces e classes

```
{ TPlacaGraficaATI }

constructor TPlacaGraficaATI.Create;
begin
  inherited;
  FMensagem := 'Os gráficos estão sendo renderizados pela placa
   ATI!'+#13#10;
end;

procedure TPlacaGraficaATI.renderizar;
begin
  ShowMessage(FMensagem);
end;

{ TPlacaGraficaNVidia }

constructor TPlacaGraficaNVidia.Create;
begin
  inherited;
  FMensagem := 'Os gráficos estão sendo renderizados pela placa
   NVIDIA!'+#13#10;
end;

procedure TPlacaGraficaNVidia.renderizar;
begin
  ShowMessage(FMensagem);
end;

{ TPlacaGraficaProprietaria }

constructor TPlacaGraficaProprietaria.Create;
begin
  inherited;
  FMensagem := 'Os gráficos estão sendo renderizados por uma placa
   Proprietaria!'+#13#10;
end;

procedure TPlacaGraficaProprietaria.renderizar;
begin
  ShowMessage(FMensagem);
end;

{ TComputador }

function TComputador.GetPlacaGrafica: IPlacaGrafica;
begin
  Result := FPlacaGrafica;
end;

constructor TComputador.Create;
begin
  inherited;
  FMensagem := 'Inicializando Computador!'+#13#10;
end;

procedure TComputador.inicializar;
begin
  if FPlacaGrafica <> nil then
  begin
    showmessage(FMensagem);
    FPlacaGrafica.renderizar;
  end
  else
    showmessage(FMensagem + 'Impossível renderizar gráficos sem
    uma placa gráfica!' + #13#10);
end;

procedure TComputador.SetPlacaGrafica(const value: IPlacaGrafica);
begin
   FPlacaGrafica := value;
end;

{ TVideogame }

function TVideogame.GetPlacaGrafica: IPlacaGrafica;
begin
  Result := FPlacaGrafica;
end;

constructor TVideogame.Create;
begin
  inherited;
  FMensagem := 'Inicializando Videogame!'+#13#10;
end;

procedure TVideogame.inicializar;
begin
  if FPlacaGrafica <> nil then
  begin
    showmessage(FMensagem);
    FPlacaGrafica.renderizar;
  end
  else
    showmessage(FMensagem + 'Impossível renderizar gráficos sem
     uma placa gráfica!' + #13#10);
end;


procedure TVideogame.SetPlacaGrafica(const value: IPlacaGrafica);
begin
  FPlacaGrafica := value;
end;
```

Listagem 11. Evento OnClick do Botão btPlacaGrafica

```
procedure TfrmPrincipal.btPlacaGraficaClick(Sender: TObject);
var
  radeon,  gforce, proprio: IPlacaGrafica;
  iComp1,iComp2,iComp3,iComp4: IComputador;
begin
  iComp1 := TComputador.Create;
  iComp2 := TComputador.Create;
  iComp3 := TComputador.Create;
  iComp4 := TVideogame.Create;

  radeon := TPlacaGraficaATI.Create;
  gforce := TPlacaGraficaNvidia.Create;
  proprio := TPlacaGraficaProprietaria.Create;

  //antes de acoplar a placa gráfica não podemos renderizar
  showmessage('=== inicializando sem placa ===');
  iComp1.inicializar;
  iComp2.inicializar;
  iComp3.inicializar;
  iComp4.inicializar;

  //acoplando as placas gráficas
  showmessage('=== acoplando as placas ===');
  iComp1.PlacaGrafica :=  radeon;
  iComp2.PlacaGrafica :=  gforce;
  iComp3.PlacaGrafica := gforce; //esta gforce terá o contador
   de referências = 2, porque dois computadores usam a mesma placa
  (impossível na vida real)
  iComp4.PlacaGrafica :=  proprio;

  //depois das placas devidamente acopladas
  showmessage('=== inicializando COM as placas ===');
  iComp1.inicializar;
  iComp2.inicializar;
  iComp3.inicializar;
  iComp4.inicializar;

  //renderizando pela propriedade PlacaGrafica
  showmessage('=== Renderizando pela propriedade PlacaGrafica ===');
  iComp1.PlacaGrafica.Renderizar;
  iComp2.PlacaGrafica.Renderizar;
  iComp3.PlacaGrafica.Renderizar;
  iComp4.PlacaGrafica.Renderizar;

  //será que o Tcomputador e o TVideogame implementam IPlacaGrafica?
   Poderemos renderizar gráficos novamente, depois de inicializado?
  showmessage('=== Renderizando com Typecast ===');
  (iComp1 as IPlacaGrafica).Renderizar;
  (iComp2 as IPlacaGrafica).Renderizar;
  (iComp3 as IPlacaGrafica).Renderizar;
  (iComp4 as IPlacaGrafica).Renderizar;
end;
```

Listagem 12. Novas interfaces e classes

```
 IRecursoCompartilhado = interface
   ['{AC3D7441-80F3-4FEC-BA17-E74333BF5FB3}']
   procedure MostraMensagem;
 end;

 IObjetoQueUsaRecurso = interface
   ['{B8D3F074-32D2-4555-B3F3-AC60EB93064C}']
   procedure UmMetodoQualquer;
   function GetRecursoCompartilhado: IRecursoCompartilhado;
   procedure SetRecursoCompartilhado(const
    value:IRecursoCompartilhado);
   property RecursoCompartilhado: IRecursoCompartilhado read
    GetRecursoCompartilhado write SetRecursoCompartilhado;
 end;

 TRecursoCompartilhado = class(TInterfacedObject, IRecursoCompartilhado)
 private
   FMensagem: string;
 public
   procedure MostraMensagem;
   constructor Create;
 end;

 TObjetoQueUsaRecurso = class (TInterfacedObject, IObjetoQueUsaRecurso,
   IRecursoCompartilhado)
 private
   FRecursoCompartilhado: IRecursoCompartilhado;
   FMensagem: string;
 public
   procedure UmMetodoQualquer;
   constructor Create;
   function GetRecursoCompartilhado: IRecursoCompartilhado;
  procedure SetRecursoCompartilhado(const value:IRecursoCompartilhado);
   property RecursoCompartilhado: IRecursoCompartilhado read
    GetRecursoCompartilhado write SetRecursoCompartilhado
     implements IRecursoCompartilhado;
 end;

...

{ TRecursoCompartilhado }

constructor TRecursoCompartilhado.Create;
begin
  inherited;
  FMensagem := 'Executando Método!'+#13#10;
end;

procedure TRecursoCompartilhado.MostraMensagem;
begin
  ShowMessage(FMensagem);
end;

{ TObjetoQueUsaRecurso }

function TObjetoQueUsaRecurso.GetRecursoCompartilhado:
IRecursoCompartilhado;
begin
  Result := FRecursoCompartilhado;
end;

constructor TObjetoQueUsaRecurso.Create;
begin
  inherited;
  FMensagem := 'TObjetoQueUsaRecurso criado!'+#13#10;
end;

procedure TObjetoQueUsaRecurso.UmMetodoQualquer;
begin

  if FRecursoCompartilhado <> nil then
  begin
    showmessage(FMensagem);
    FRecursoCompartilhado.MostraMensagem;
  end
  else
    showmessage(FMensagem + 'Recurso não encontrado!' + #13#10);
end;

procedure TObjetoQueUsaRecurso.SetRecursoCompartilhado
 (const  value: IRecursoCompartilhado);
begin
  FRecursoCompartilhado := value;
end;
```

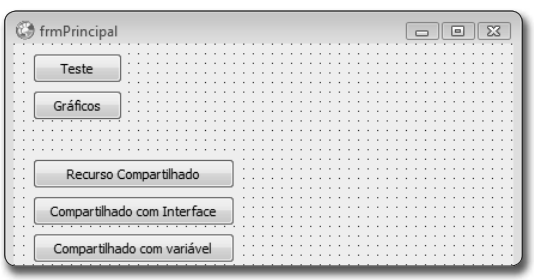


Figura 1. Botões do form principal

Você pode aplicar o mesmo teste que fizemos para testar nossos computadores e placas. Veja a **Listagem 13**, que é o evento OnClick do btRecComp.

O ponto que vamos considerar é a diferença entre usar apenas interfaces e usar variáveis. Temos que entender o problema que acontece quando misturamos tudo isso. Veja o trecho de código da **Listagem 13** onde atribuímos o recurso:

```
obj1.RecursoCompartilhado :=  rec;
obj2.RecursoCompartilhado :=  rec;
```

Agora imagine se depois que você liberasse obj1 e obj2 da memória, ainda precisasse do recurso Rec para alguma coisa. O que aconteceria? Os eventos OnClick dos botões btCompInt e btCompVar da **Listagem 14** mostram o que pode acontecer.

Observe que no evento btComVarClick a variável rec é do tipo TRecursoCompartilhado, ou seja, é uma classe e não uma interface. Então nós instanciamos todos os objetos e passamos a referência de rec para os objetos obj1 e obj2. Depois de utilizarmos esses dois objetos, nós atribuímos NIL primeiro para o obj1, que resulta na diminuição do contador de referência de rec. Sim, rec possui um contador de referências porque ele implementa uma interface. Depois atribuímos NIL para obj2. Isso faz com que novamente o contador de referências seja diminuído e reduzido ao valor 0 (zero), indicando que deve ser destruído da memória e assim é feito. Aí nesse momento tentamos executar o objeto rec e como o objeto foi liberado quando seu contador chegou a zero, é disparada uma exceção do tipo Access Violation, porque ele não existe mais. Reforço que isso aconteceu porque misturamos interfaces e objetos. Vejamos agora o evento btCompInt. Note que o objeto rec agora é do tipo IRecursoCompartilhado, ou seja, é uma interface. O código de utilização é praticamente o mesmo só que desta vez não há Access Violation. Isso se deve ao fato que, quando utilizamos uma interface para armazenar uma referência, ao instanciarmos o objeto que implementa a interface, seu contador já é acrescido. Ou seja, quando executamos:

```
rec := TRecursoCompartilhado.Create;
```

O contator de rec passa a valer um, o que não aconteceria se rec fosse do tipo TRecursoCompartilhado. Sendo assim, ao mantermos rec como interface, após o associarmos aos objetos obj1 e obj2, seu contador estará valendo 3. Por isso mesmo depois de liberar obj1 e obj2 da memória ainda é possível utilizar o rec, porque ainda existe uma referência, ele mesmo!

É possível obter com precisão o número de referências de um objeto que implemente uma interface. Vamos fazer um ajuste para mostrarmos isso. Crie um descendente de TInterfacedObject que mostre o FRefCount (número de referências) nos métodos _AddRef e _Release, e faça com que a classe TRecursoCompartilhado seja descendente dele e não de TInterfacedObject, como mostra a **Listagem 15**.

Os métodos _AddRef e _Release são responsáveis, respectivamente, pela adição e subtração na contagem de referência. Estão implementados em TInterfacedObject na seção protected, porém como criamos uma classe que herda de TInterfacedObject, seus membros protegidos se tornam acessíveis e assim é podemos sobrecarregá-los para que, ao serem executados seja exibida uma mensagem que exibe o total de referências.

Agora que aprendemos a usar bem as interfaces e delegações sem causar erros ou leaks, vamos conhecer outras classes do Delphi para se trabalhar com interfaces.

Listagem 13. Evento OnClick do btRecComp

```
procedure TfrmPrincipal.btRecCompClick(Sender: TObject);
var
  rec: IRecursoCompartilhado;
  obj1,obj2: IObjetoQueUsaRecurso;
begin
  obj1 := TObjetoQueUsaRecurso.Create;
  obj2 := TObjetoQueUsaRecurso.Create;
  rec := TRecursoCompartilhado.Create;

  showmessage('=== executando método sem o recurso ===');
  obj1.UmMetodoQualquer;
  obj2.UmMetodoQualquer;

  showmessage('=== setando os recursos ===');
  obj1.RecursoCompartilhado :=  rec;
  obj2.RecursoCompartilhado :=  rec;

  showmessage('=== executando método COM o recurso ===');
  obj1.UmMetodoQualquer;
  obj2.UmMetodoQualquer;

  showmessage('=== usando propriedade RecursoCompartilhado ===');
  obj1.RecursoCompartilhado.MostraMensagem;
  obj2.RecursoCompartilhado.MostraMensagem;

  showmessage('=== usando typecast ===');
  (obj1 as IRecursoCompartilhado).MostraMensagem;
  (obj2 as IRecursoCompartilhado).MostraMensagem;
end;
```

Listagem 14. Botões btCompVar e btCompInt

```
procedure TfrmPrincipal.btCompVarClick(Sender: TObject);
var
  rec: TRecursoCompartilhado;
  obj1,obj2: IObjetoQueUsaRecurso;
begin
  obj1 := TObjetoQueUsaRecurso.Create;
  obj2 := TObjetoQueUsaRecurso.Create;
  rec := TRecursoCompartilhado.Create;

  obj1.RecursoCompartilhado :=  rec;
  obj2.RecursoCompartilhado :=  rec;

  obj1.UmMetodoQualquer;
  //destruo a referencia ao obj1
  obj1 := nil;
  //isso diminuirá em 1 do contador de referencias de rec,
  //mas mesmo assim ele ainda existe com 1 referencia, em 1 obj2

  obj2.UmMetodoQualquer;
  //diminuimos mais uma referencia, e agora, o que acontece com rec?
  obj2 := nil;

  rec.MostraMensagem;
  //isso vai dar erro, pode ser access violation, invalid
   pointer operation ou simplesmente um erro de lógica
end;

procedure TfrmPrincipal.btCompIntClick(Sender: TObject);
var
  rec: IRecursoCompartilhado;
  obj1,obj2: IObjetoQueUsaRecurso;
begin
  obj1 := TObjetoQueUsaRecurso.Create;
  obj2 := TObjetoQueUsaRecurso.Create;
  rec := TRecursoCompartilhado.Create;

  obj1.RecursoCompartilhado :=  rec;
  obj2.RecursoCompartilhado :=  rec;

  obj1.UmMetodoQualquer;
  //destruo a referencia ao obj1
  obj1 := nil;
  //isso diminuirá em 1 do contador de referencias de rec,
  //mas mesmo assim ele ainda existe com 2 referencias,
  //pois tem 1 obj2 e 1 em rec

  obj2.UmMetodoQualquer;
  //diminuimos mais uma referencia, agora eu só tenho rec
  obj2 := nil;

  rec.MostraMensagem;
  //posso usar rec normalmente aqui,
  //e depois atribuir nil a ele ou não

end;
```

Listagem 15. Classe para monitorar a contagem de referências de uma interface

```
TContadorInterface = class(TInterfacedObject)
 public
   function _AddRef: Integer; stdcall;
   function _Release: Integer; stdcall;
end;


TRecursoCompartilhado = class(TContadorInterface,
 IRecursoCompartilhado)
 private
   FMensagem: string;
 public
   procedure MostraMensagem;
   constructor Create;
 end;

...


{ TContadorInterface }

function TContadorInterface._AddRef: Integer;
begin
  Result := inherited  _AddRef;
  ShowMessage(IntToStr(FRefCount));
end;

function TContadorInterface._Release: Integer;
begin
  result := inherited _Release;
  ShowMessage(IntToStr(FRefCount));
end;
```

Mais recursos para interfaces

O nosso computador hipotético esperava que alguém plugasse uma placa de vídeo para depois renderizar os gráficos. Mas nenhum computador inicializa sem a placa de vídeo e ainda espera que você a plugue com ele ligado, certo? Ele já deve inicializar com a placa. Para implementar isso vamos conhecer as classes TAggregatedObject e TContainedObject e as diferenças entre elas.

A primeira coisa que veremos é que quando usamos esses tipos de objetos, o recurso compartilhado será passado como parâmetro para o construtor deles, e será uma propriedade somente leitura, não podendo ser alterado depois. Outra coisa importante: TContainedObject é filho de TAggregatedObject.

O mais importante é que quando usamos estas classes, usamos sua instância em uma variável do tipo objeto, mas seu recurso compartilhado, chamado aqui de controller, é uma interface. Precisamos criar a classe TAggregatedObject e seus descendentes em uma variável do tipo objeto para poder destruir depois. Essas classes nós somos obrigados a destruir, porque o contador de referências delas são diferentes, eles apenas decrementam as referências do seu controller, mas sem fazer um controle próprio de referências ou se autodestruir quando o número de referências a ele ou ao controller chegar a zero.

Uma maneira de contornar isso é desenvolver seu próprio mecanismo de contagem de referência interno sobrecarregando os métodos _AddRef e _Release, mas isso foge ao escopo deste artigo.

Vamos alterar nosso TObjetoQueUsaRecurso para que ele seja descendente de TAggregatedObject e alterar o get e set para que usem seu objeto controller. Para isso vejamos a **Listagem 16.**

Observe que agora o construtor da classe TObjetoQueUsaRecurso requer a passagem de um parâmetro do tipo interface, que fica armazenado em Controller . Posteriormente os Métodos GetRecursoCompartilhado, SetRecursoCompatilhado e UmMetodoQualquer fazem uso dele exclusivamente. Poderíamos fazer isso com o nosso TComputador, TVideogame etc., preferimos fazer com o TObjetoQueUsaRecurso para facilitar o entendimento.

Vamos executar agora os testes dos botões btRecComp, btCompVar, btCompInt, seguindo a mesma lógica que seguimos até agora, conforme a **Listagem 17**. O objeto

Listagem 16. Alterações na interface IObjetoQueUsaRecurso

```
IRecursoCompartilhado = interface
  ['{AC3D7441-80F3-4FEC-BA17-E74333BF5FB3}']
  procedure MostraMensagem;
end;

IObjetoQueUsaRecurso = interface
  ['{B8D3F074-32D2-4555-B3F3-AC60EB93064C}']
  procedure UmMetodoQualquer;
  function GetRecursoCompartilhado: IRecursoCompartilhado;
  property RecursoCompartilhado: IRecursoCompartilhado read
   GetRecursoCompartilhado;
end;

TRecursoCompartilhado = class(TContadorInterface,
 IRecursoCompartilhado)
private
  FMensagem: string;
public
  procedure MostraMensagem;
  constructor Create;
end;

TObjetoQueUsaRecurso = class (TAggregatedObject,
IObjetoQueUsaRecurso, IRecursoCompartilhado)
private
  FMensagem: string;
public
  procedure UmMetodoQualquer;
  constructor Create(const Controller: IInterface);
  function GetRecursoCompartilhado: IRecursoCompartilhado;
  property RecursoCompartilhado: IRecursoCompartilhado read
   GetRecursoCompartilhado implements IRecursoCompartilhado;
end;

...

{ TRecursoCompartilhado }

constructor TRecursoCompartilhado.Create;
begin
  inherited;
  FMensagem := 'Executando Método!'+#13#10;
end;

procedure TRecursoCompartilhado.MostraMensagem;
begin
  ShowMessage(FMensagem);
end;

{ TObjetoQueUsaRecurso }

function TObjetoQueUsaRecurso.GetRecursoCompartilhado:
IRecursoCompartilhado;
begin
  Result := Controller as IRecursoCompartilhado;
end;

constructor TObjetoQueUsaRecurso.Create(const Controller:
IInterface);
begin
  inherited Create(Controller);
  FMensagem := 'TObjetoQueUsaRecurso criado!'+#13#10;
end;

procedure TObjetoQueUsaRecurso.UmMetodoQualquer;
begin

  if Controller <> nil then
  begin
    showmessage(FMensagem);
    (Controller as IRecursoCompartilhado).MostraMensagem;
  end
  else
    showmessage(FMensagem + 'Recurso não encontrado!' + #13#10);
end;
```

TRecursoCompartilhado continuará sendo descendente de TInterfacedObject ou de TContadorInterface quando quisermos debugar a contagem de referências.

Listagem 17. Novo código para os botões

```
procedure TfrmPrincipal.btRecCompClick(Sender: TObject);
var
  rec: IRecursoCompartilhado;
  obj1,obj2: TObjetoQueUsaRecurso;
begin
  rec := TRecursoCompartilhado.Create;
  obj1 := TObjetoQueUsaRecurso.Create(rec);
  obj2 := TObjetoQueUsaRecurso.Create(rec);

  showmessage('=== executando método COM o recurso ===');
  obj1.UmMetodoQualquer;
  obj2.UmMetodoQualquer;

  showmessage('=== usando propriedade RecursoCompartilhado ===');
  obj1.RecursoCompartilhado.MostraMensagem;
  obj2.RecursoCompartilhado.MostraMensagem;

  showmessage('=== usando typecast ===');
  (obj1 as IRecursoCompartilhado).MostraMensagem;
  (obj2 as IRecursoCompartilhado).MostraMensagem;

  obj1.Free;
  obj2.Free;
end;

procedure TfrmPrincipal.btCompVarClick(Sender: TObject);
var
  rec: TRecursoCompartilhado;
  obj1,obj2: TObjetoQueUsaRecurso;
begin
  rec := TRecursoCompartilhado.Create;
  obj1 := TObjetoQueUsaRecurso.Create(rec);
  obj2 := TObjetoQueUsaRecurso.Create(rec);

  obj1.UmMetodoQualquer;
  obj2.UmMetodoQualquer;
  rec.MostraMensagem;

  obj1.Free;
  obj2.Free;
end;

procedure TfrmPrincipal.btCompIntClick(Sender: TObject);
var
  rec: IRecursoCompartilhado;
  obj1,obj2: TObjetoQueUsaRecurso;
begin
  rec := TRecursoCompartilhado.Create;
  obj1 := TObjetoQueUsaRecurso.Create(rec);
  obj2 := TObjetoQueUsaRecurso.Create(rec);

  obj1.UmMetodoQualquer;
  obj2.UmMetodoQualquer;
  rec.MostraMensagem;

  obj1.Free;
  obj2.Free;
end;
```

Tente compilar, não vai conseguir, pois ocorrerão erros nas linhas:

```
showmessage('=== usando typecast ===');
(obj1 as IRecursoCompartilhado).MostraMensagem;
(obj2 as IRecursoCompartilhado).MostraMensagem;
```

Comente-as por enquanto.

Executando o teste você verá que não ocorrem leaks. Nos botões btCompInt e btCompVar você notará novamente que a mistura entre variáveis e interfaces pode causar problemas na contagem de referências, causando access violations e outras consequências embaraçosas. Então, btCompVar é o exemplo máximo do que NÃO FAZER.

Agora vamos remover o comentário daquelas 3 linhas que comentamos e trocar o pai de TObjetoQueUsaRecurso para TContainedObject, conforme a linha a seguir:

```
TObjetoQueUsaRecurso = class (TContainedObject,
IObjetoQueUsaRecurso, IRecursoCompartilhado)
```

Compile e teste. Como você pode perceber esta é a diferença entre o TContainedObject e o TAggregatedObject: Por ter o método QueryInterface diferenciado, o TContainedObject permite fazer typecast para a interface do recurso compartilhado, do objeto que ele contém, usando o operador **as.** Enquanto que o TAggregatedObject não permite esse tipo de typecast. O TAggregatedObject delega a implementação de uma interface a um controller interno, mas não deixa isso explícito como o TContainedObject.

Conclusão

Delegações e aggregates são recursos poderosos do Delphi para deixar suas aplicações mais robustas e flexíveis. São recursos muito usados, por exemplo, em bibliotecas de componentes conhecidas. Mesmo assim poucos desenvolvedores os utilizam no seu dia-a-dia. Ainda alguns conhecem esses recursos, mas não sabem a maneira correta de utilizá-los, causando erros ou leaks nas aplicações.

Esperamos que com a leitura deste artigo o leitor adquira conhecimentos básicos sobre os recursos em questão. Trabalhar usando interfaces além de facilitar a compreensão do todo em POO, também facilita o desenvolvimento em equipe e a aplicação de metodologias ágeis de desenvolvimento. Usar delegações permite criar regras específicas quanto a ligações entre objetos, funcionalidades de cada um, quem implementa o que, separação de conceitos e encapsulamento. ●

Links

Delphi reference counted interfaces
http://edn.embarcadero.com/article/30125

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

A partir de agora você pode contar com a ajuda da

Chegou a Consultoria On-line DevMedia

Consultoria Técnica + Professor Virtual + Certificação

Mais Informações:
www.devmedia.com.br/consultoria_online (21) 3382-5038

DevMedia em seus projetos e estudos.

A DevMedia possui um numeroso time de autores, editores e professores que juntos produzem o material que você está acostumado a encontrar em nosso site e revistas. E são exatamente esses mesmos profissionais que estarão a sua disposição para tirar suas dúvidas e ajudá-lo em seus projetos e estudos. Através de uma plataforma 100% web a Consultoria DevMedia garante sigilo absoluto, eficiência e rapidez em todas as respostas. Finalmente você terá ao seu alcance uma consultoria de qualidade por um preço muito acessível. Consulte nossos planos.

Mais um serviço DevMedia group

Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

DBX e DataSnap

O que há de novo no acesso a dados no Delphi 2010

Do que trata o artigo

Ilustrar todo o poder e evolução da tecnologia DataSnap, que desde sua criação, tem passado por grandes transformações, tornando-se uma das principais tecnologias para o desenvolvimento de alta perfomance em ambientes multicamadas. Justificar a utilização do DBX em qualquer tipo de aplicação, exibindo agora, além dos novos recursos, a grande e esperada novidade do driver nativo de conexão com o Firebird.

Para que serve

Criação de aplicações multicamadas (distribuídas), permitindo o desenvolvimento adotando o máximo de performance e segurança utilizando os novos e melhorados recursos do DBXFramework, agora com driver 100% compatível com o banco de dados Firebird.

Em que situação o tema é útil

Os diversos recursos apresentados podem ser utilizados para a criação e integração com outras linguagens e tecnologias, melhorias na atual arquitetura de suas aplicações, criação de soluções mais robustas, inteligentes, profissionais, exibição de novas técnicas para aplicações em ambientes multicamadas. Estes recursos são muito utilizados em desenvolvimento de aplicações de missão crítica como Home Banking, Transportadoras, Sistemas financeiros, integração entre grandes empresas etc.

Daniel Sobrinho Laporte
daniel.laporte@gmail.com
daniel.sobrinho@devmedia.com.br)
Editor técnico da revista ClubeDelphi. Graduado em Sistemas de informação e pós-graduado em administração de bancos de dados pela Universidade de Franca. É analista de sistemas e desenvolvedor Delphi desde 2005 desenvolvendo aplicações cliente/servidor e multicamadas em Delphi e Firebird.

Resumo do DevMan

Com o lançamento da nova versão do Delphi, estamos cheios de novidades, curiosidades e melhorias. Neste artigo iremos conhecer algumas novidades da nova versão do RAD Studio, focando principalmente nas melhorias implementadas no DBX e novo DataSnap. Acompanhe agora um artigo sobre algumas das novidades da versão que promete ser a melhor de todas já feitas.

Desde a venda da CodeGear, a Embarcadero tem se preocupado em criar soluções cada vez mais robustas e integradas a todos os tipos de tecnologias. Com a chegada do RAD Studio 2009, tivemos também o lançamento do Delphi Prism que nos permite desfrutar de todo o poder do .NET Framework nativamente em nossas aplicações. E é claro que este é um fato extremamente positivo, pois além do Delphi for PHP, tivemos mais

uma ferramenta para possibilitar o desenvolvimento em outras tecnologias. Perceba que com todos os lançamentos que tivemos, fomos capazes de expandir nossos horizontes, pois, agora é possível utilizarmos Delphi para criarmos soluções Win32, utilizar o ambiente (IDE) Delphi para desenvolvermos soluções de alto nível para WEB em PHP, ou mesmo adotar .NET no ambiente do Visual Studio para criarmos projetos com o que há de mais novo, sendo aplicações Win32 WEB, Mobile dentre muitas outras. Com o lançamento do RAD Studio 2010, inúmeros recursos foram criados e novamente nossos horizontes foram expandidos. Desde a famosa versão 7, o Delphi tem evoluído cada vez mais. A palavra da vez agora com certeza é interatividade. Nunca tivemos uma diversidade tão grande de tecnologias com possibilidade de interação. Com chegada da nova versão isto não foi diferente, pois agora é possível utilizarmos, comunicarmos e disponibilizarmos métodos via JSON e REST, o que expande ainda mais nossa capacidade de interação com outras linguagens.

Se atentarmos bem à nova solução RAD, o número de novidades é de quase 120 (confira mais detalhes na seção links no final do artigo), RTTI avançada, suporte a Gesture e Touch, IDE Insight, novas Units, novos componentes, suporte a filtros de compressão de dados, tolerância à falhas, balanceamento de carga, suporte a HTTP, callbacks e muito mais. Em especial, gostaria de destacar, as grandes melhorias e evolução do DBX e DataSnap. A comunidade de desenvolvedores Delphi-Firebird finalmente ganhou o tão esperado driver de conexão nativa, que possibilita utilização do banco de dados com performance e sem necessidade de adquirirmos drivers ou componentes de terceiros que nos faziam abdicar de todo o poder do novo DBX Framework.

Durante o artigo iremos conhecer algumas das novidades elaborando pequenos exemplos para abordar os novos recursos de uma maneira simples e aplicável a seu dia-a-dia. Portanto, criaremos um banco de dados e uma mini-aplicação cliente-servidor que simule um cadastro real de uma aplicação e reaproveitaremos nosso banco de dados para criarmos uma solução multicamadas. Mãos à obra!

Novidades do dbExpress

Com certeza absoluta, uma das novidades que mais agradou a comunidade de desenvolvedores foi a criação do novo driver nativo de conexão com o Firebird. Desde a versão 7 do Delphi nós temos adotado inúmeras soluções para conseguirmos criar aplicações com esta dupla. Quem não se lembra de quando havia apenas as versões 1.0 e 1.5 do Firebird? Nesta época, realizávamos a conexão através do DBX 3 utilizando o driver de conexão com o banco de dados Interbase, ou caso contrário, adotávamos a famosa dbexpUIBfire15.dll, que nos permitia o acesso nativo à versão 1.5 do Firebird de forma mais rápida. A utilização do Firebird com o driver de conexão do Interbase (dbexpint. dlll) era nativa devido ao fato de que ambas ainda possuíam estruturas semelhantes, pois como todos nós sabemos, o Firebird nasceu a partir do Interbase.

Com a evolução destes dois bancos e com a própria evolução do Delphi, estes foram ganhando características próprias. Com a aquisição da linha de desenvolvimento da Borland pela CodeGear, tivemos a criação do Delphi 2007, e com isto, o nosso DBX foi reescrito dando origem a sua versão 4. Ainda é possível conectar-se a uma base de dados Firebird por meio do driver do Interbase, que utiliza-se da dbxint.dll, porém devido às evoluções do Framework, do Interbase e das versões 2.x do Firebird, a compatibilidade total foi comprometida, desta forma tínhamos o problema de incompatibilidade com campos Blob por exemplo. Com isso, surgiram soluções proprietárias e até mesmo gratuitas para sanar este problema. Podemos citar os drivers da Corelab (solução proprietária). Devido aos fatores de evolução do DBX e variedade de drivers proprietários, muitos desenvolvedores optaram por não utilizar o Framework, adotando outras soluções como o ZeosDBO, componentes MDO, IBObjects, FIBPlus, IBDac entre muitas outras.

Com o lançamento do Delphi 2010 temos conexão nativa com Firebird nas versões 1.5 e 2.x. E mais, não foi só esta a novidade. Nosso Framework agora está muito mais poderoso, além de nos permitir total integração com Interbase 2009, SQL Server 2008, MySQL 5.1 e Oracle 11g. Novamente, nossa gama de variedades de conexão com diferentes bancos de dados foi expandida, com isto, podemos desenvolver aplicações multibanco com altíssima performance, qualidade, robustez e o melhor, sem grandes preocupações com adaptações de incompatibilidade.

Outro recurso que foi implementado é relativo ao formato de campos do tipo Date/Time. Agora o formato que deve ser adotado para cada tipo de base de dados é indiferente, pois o próprio Framework já sabe como interpretar e enviar os dados no formato correto. Perceba que temos muitas novidades, e este é só o começo.

Mais adiante iremos apresentar outras funcionalidades no que diz respeito ao DataSnap, mas como estamos agora abordando o DBExpress, faremos uma aplicação de exemplo, conectando com o banco de dados, e efetuando um cadastro. No decorrer deste artigo iremos realizar novos exemplos como já citado anteriormente, para que com isto possamos demonstrar mais alguns novos recursos que ainda serão abordados. Vamos ao exemplo.

Criando o banco de dados

Para não prolongarmos muito, iremos criar apenas um cadastro de clientes, porém, para simular um dos principais problemas que tínhamos na versão do Delphi 2007, definiremos em nossa estrutura campos BLOB de ambos os tipos (texto e binário) e faremos com que sejam inseridas imagens em nosso cadastro. Como podemos observar, o código para a criação do banco de dados é ilustrado na **Listagem 1**. Para sua criação utilize uma ferramenta de seu gosto. Para o exemplo utilizei o IBExpert (o link para download está na sessão links).

Criando a aplicação

Como já vimos em outras matérias, a partir da versão do Delphi 2009 a conexão do DBExpress com o banco de dados

deve ser configurada no Data Explorer (e não mais com o duplo clique sobre o SQLConnection), que pode ser facilmente acessado pelo menu View>Data Explorer. Na janela do Data Explorer perceba que agora temos a opção de criar uma conexão com o Firebird, como mostra a **Figura 1.**

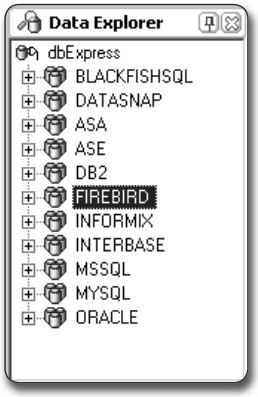


Figura 1. Opção de conexão com Firebird

Clique sobre a conexão com o botão direito e selecione Add New Connection para que a janela de Wizard seja exibida. Informe ClubeDelphi em Connection Name, em seguida faça a expansão da árvore de conexão selecionando a nossa conexão criada. Com o botão direito, selecione a opção Modify Connection. Na nova janela selecione a opção Advanced e configure como ilustra a **Figura 2.** Após configurar, dê OK e faça um Test Connection. Seu resultado será o da **Figura 3.**

Listagem 1. Script do banco de dados

```
SET SQL DIALECT 3;
SET NAMES WIN1252;

SET CLIENTLIB '<DIRETÓRIO DA INSTALAÇÃO DO FIREBIRD>\BIN\
fbclient.dll';

CREATE DATABASE '<DIRETÓRIO DO BANCO DE DADOS>\DELPHI2010.FDB'
USER 'SYSDBA' PASSWORD 'masterkey'
PAGE_SIZE 4096

DEFAULT CHARACTER SET WIN1252;

CREATE TABLE CLIENTE
(
    IDCLIENTE  INTEGER NOT NULL,
    CLIENTE    VARCHAR(50),
    DATA_NASCIMENTO DATE,
    ENDERECO   VARCHAR(50),
    NUMERO     INTEGER,
    CIDADE     VARCHAR(50),
    BAIRRO     VARCHAR(50),
    ESTADO     CHAR(2),
    FOTO       BLOB SUB_TYPE 0 SEGMENT SIZE 80 ,
    OBSERVACAO BLOB SUB_TYPE 1 SEGMENT SIZE 80
);

ALTER TABLE CLIENTE
ADD CONSTRAINT PK_CLIENTE
PRIMARY KEY (IDCLIENTE)
USING INDEX IDX_CLIENTE;

CREATE SEQUENCE GEN_CLIENTE_ID;

CREATE OR ALTER TRIGGER CLIENTE_BI FOR CLIENTE
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.IDCLIENTE IS NULL) THEN
    NEW.IDCLIENTE = GEN_ID(GEN_CLIENTE_ID,1);
END;
```

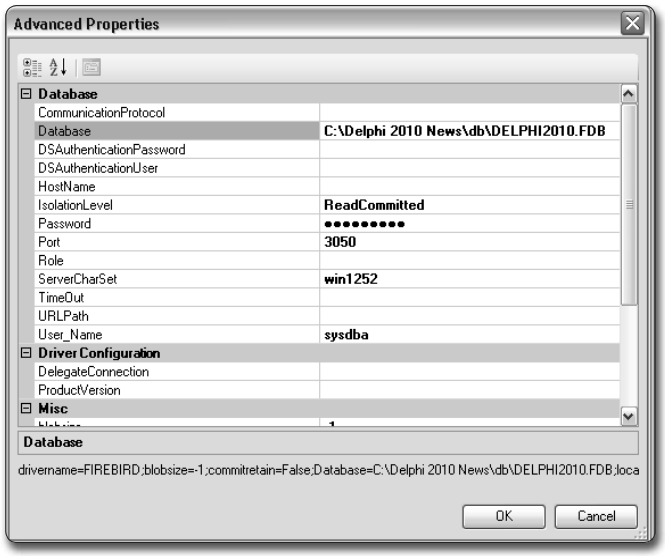


Figura 2. Configuração da conexão com o banco

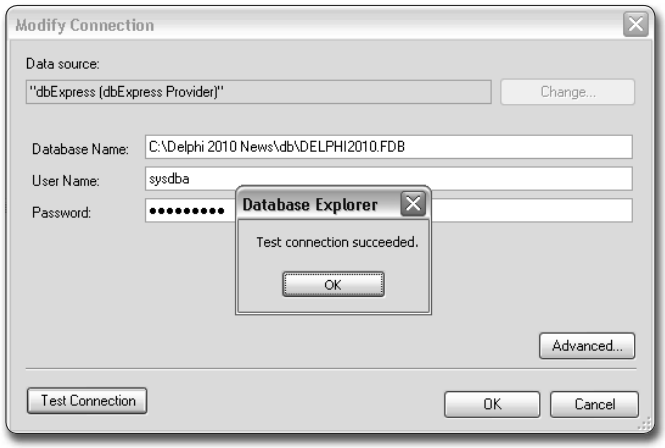


Figura 3. Conexão efetuada com sucesso

Crie um novo projeto VCL Forms Application. Selecione no Data Explorer a opção Tables e perceba que logo aparecerá a tabela CLIENTE. Arraste a tabela para o formulário e note que automaticamente foi criado um SQLConnection com o nome de nossa conexão e um SQLDataSet com o nome da tabela selecionada. Atente para a .dll que está sendo utilizada na propriedade LibraryName do SQLConnection conforme mostra a **Figura 4.**

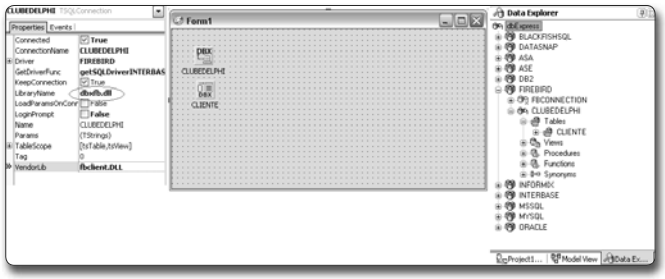


Figura 4. Nova .dll de conexão com o Firebird

Após ter realizado o teste de conexão, vamos adicionar alguns componentes para simular uma aplicação. Insira no

mesmo formulário um DataSetProvider, um ClientDataSet, um DataSource, um DBNavigator, um Button e os demais campos relativos aos Fields. Renomeie os componentes respectivamente para dspCLIENTE, cdsCLIENTE, dsrCLIENTE e btnImagem. Dê um duplo clique sobre o SQLDataSet (CLIENTE) e no Fields Editor selecione Add All Fields, configurando os ProviderFlags do campo IDCLIENTE para [pfInkey, pfInUpdate, pfInWhere] e desative a propriedade Required marcando-a para False para que o valor do campo ID seja gerado pela nossa Trigger, para os demais campos deixe apenas o Provider Flag [pfInUpdate]. No DataSetProvider, aponte à propriedade DataSet o SQLDataSet e modifique seu UpdateMode para upWhereKeyOnly, por último, adicione ao Options as funcionalidades [poIncFieldProps, poPropagateChanges]. No ClientDataSet indique o dspCLIENTE na propriedade ProviderName. Aponte o DataSource para o cdsCLIENTE. Por último, ligue o DBNavigator e os demais componentes ao DataSource através da propriedade de mesmo nome. Feitas as configurações, no evento onCreate do formulário realize a abertura da conexão com o banco de dados conforme mostrado a seguir:

```
cdsCLIENTE.Close;
cdsCLIENTE.Open;
```

Nos eventos AfterPost e AfterDelete do cdsCLIENTE simplesmente aplique as alterações no banco de dados através do comando ApplyUpdates:

```
cdsCLIENTE.ApplyUpdates(0);
```

Selecione o botão de imagem e no evento OnClick, implemente o código para a seleção de imagens conforme mostrado na **Listagem 2**.

Listagem 2. Evento OnClick

```
var
//Adicione ExtDlgs na cláusula uses
  dlg : TOpenPictureDialog;
begin
  if (cdsCLIENTE.Active) and (cdsCLIENTE.State in
   [dsEdit, dsInsert]) then
  begin
    try
      dlg := TOpenPictureDialog.Create(Self);
      dlg.Filter := 'Arquivos de imagens(*.bmp; *.png;) | *.bmp;
       *.png;';
      dlg.Execute;
      if Trim(dlg.FileName) <> '' then
        cdsCLIENTEFOTO.LoadFromFile(dlg.FileName);
    finally
      FreeAndNil(dlg);
    end;
  end;
```

Como podemos observar, o código é bem simples. Criamos um OpenPictureDialog (dlg) em runtime para buscar as imagens. Verificamos se o DataSet está ativo e em estados de inserção/edição. Sendo satisfeita nossa condição, realizamos um filtro de imagens .bmp e .png e executamos, finalizamos verificando se foi informada uma imagem, atribuindo em seguida o conteúdo ao campo BLOB e destruindo o componente criado em tempo de execução. Veja a interface de nossa mini-aplicação na **Figura 5**.

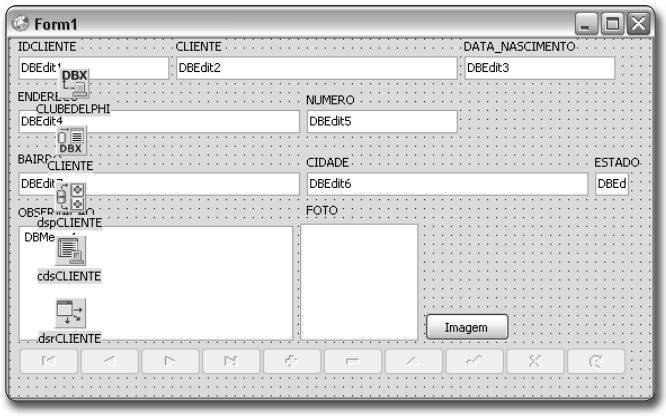


Figura 5. Interface da aplicação

Execute nossa aplicação e veja que todas as alterações já estão sendo aplicadas no banco de dados normalmente, e o mais importante, de forma nativa, possibilitando maior performance, observe também que conseguimos gravar os campos BLOB de ambos os tipos sem dificuldade alguma. Salve sua aplicação nomeando a unit principal como untPrincipal.pas e o seu projeto como FBApp.dproj. Passaremos agora as outras novidades da versão, falando um pouco de DataSnap.

Novidades do DataSnap

Nas versões passadas, o atual DataSnap era chamado de Midas como sabemos. Com o passar do tempo e evolução, o Framework passou por várias modificações, sendo cada vez mais melhorado e ganhando novas funcionalidades.

Para quem já trabalhou com as versões passadas do Delphi, deve lembrar das diferentes tecnologias para criação de aplicações multicamadas. Passado o tempo, utilizamos DCOM, COM+ e BSS, mas o Framework que sempre esteve presente em todas estas situações de desenvolvimento foi o DataSnap. Com a chegada da versão RAD 2009, tivemos uma inovação na maneira que estávamos habituados a criar aplicações N-tier, contando com a introdução dos Server Modules, novos componentes DataSnap, integração com o DBX Framework, criação de aplicações .NET adotando o Delphi Prism, disponibilização dos métodos do servidor de aplicação para a camada cliente por meio da classe Proxy gerada pelo Framework do DBX dentre muitas outras novidades. Com tudo isso passamos a nos referir carinhosamente a este novo modelo como DataSnap 2009. Na nova versão 2010 tivemos inúmeras melhorias, possibilidade de integração com novas tecnologias, wizards de acompanhamento para a criação das nossas aplicações multicamadas, implementação de Tunelling de dados, disponibilização dos métodos via WEB por meio de JSON/REST, integração com o SOAP 1.2, Callbacks, melhorias na parte de segurança, capacidade de realização de Load Ballance, Fail-Over, suporte a TCP/IP e HTTP, filtros de compressão de dados com ZLib e muito mais. Exibiremos alguns destes recursos criando uma aplicação multicamadas desde o princípio e aplicando os principais recursos à mesma.

Nota

Vale ressaltar que agora é possível termos acesso ao código fonte da Midas.dll, que nada mais é do que a principal .dll responsável pelo DataSnap. Com isso, podemos modificá-la e até mesmo estudá-la à vontade. Lembrando que esta foi escrita em linguagem C.

Novos wizards para criação de aplicações n-tier

Na versão passada, havia duas maneiras de criarmos uma aplicação servidora com o novo modelo. A primeira consistia em criar uma aplicação a qual necessitava ser iniciada manualmente. A segunda abordava a criação de um serviço, para inicialização automática. O fato é que, em qualquer uma das duas situações, era necessário criarmos tudo manualmente, Server Modules, Container (Data Module) e até mesmo o serviço. Na nova versão, temos agora a ajuda de um Wizard que nos auxilia muito na criação de uma nova aplicação, permitindo inclusive a criação do serviço automático, com possibilidade de autenticação via protocolo HTTP. Iremos dar início a um pequeno exemplo de aplicação multicamadas utilizando nosso banco de dados já existente. Então acesse o menu File>New>Other e veja que do lado esquerdo temos a opção DataSnap Server, enquanto do lado direito temos as opções DataSnap Server, DataSnap WebBroker Application e Server Module (**Figura 6**).

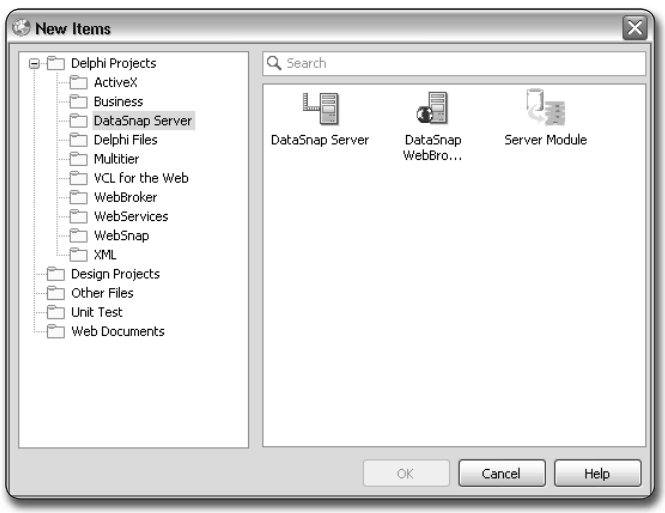


Figura 6. Opção para criação de um DataSnap Server

Após ter selecionado DataSnap Server e clicado em OK será exibida uma nova janela, configure a sua como a da **Figura 7**.

Repare que ao término de sua criação será criado automaticamente um Container, um Server Module contendo todos os componentes já previamente configurados e um formulário para exibição do servidor de aplicação. Note que no evento OnGetClass do componente DSServerClass já foi realizado o registro do Server Module, permitindo a visualização na aplicação cliente. Renomeie o novo formulário para frmPrincipal, alterando seu Caption para AppServer, no Server Module modifique a propriedade Name para ServerModulePrincipal e adicione um SQLConnection, um SQLDataSet e um DataSetProvider. Nomeie-os para sqlConexao, sqlCliente e dspCliente respectivamente configurando-os como no exemplo anterior. Altere a propriedade CommandText do sqlCliente como na **Listagem 3.**

Figura 7. Etapa final de criação do DataSnap Server

Listagem 3. CommandText do sqlCliente

```
SELECT
  CLIENTE.IDCLIENTE,
  CLIENTE.CLIENTE,
  CLIENTE.DATA_NASCIMENTO,
  CLIENTE.ENDERECO,
  CLIENTE.NUMERO,
  CLIENTE.BAIRRO,
  CLIENTE.CIDADE,
  CLIENTE.ESTADO,
  CLIENTE.OBSERVACAO,
  CLIENTE.FOTO
FROM
  CLIENTE
WHERE
  CLIENTE.IDCLIENTE =:IDCLIENTE
```

No Server Container mantenha os mesmos nomes dos componentes, removendo apenas o número 1 de cada um deles. Adicione ao Uses a nova unit referente ao Server Module, remova a unit inserida automaticamente pelo Delphi e modifique o evento OnGetClass para:

```
PersistentClass := TServerModulePrincipal;
```

Nosso servidor de aplicação está pronto em poucos minutos e sem grandes esforços. Salve a aplicação nomeando as units para untServerModulePrincipal, untPrincipal e untServerCointainer e o projeto principal para AppServer.dproj. Veremos agora duas novidades da nova versão.

Abra o seu Server Container e repare que temos dois novos componentes (DSHTTPService e DSHTTPServiceAuthenticationManager). Com esta nova dupla é possível a disponibilização dos nossos métodos na WEB, através de REST e dos comandos POST, GET, PUT e DELETE, permitindo-os

serem exibidos por outras tecnologias, bem como consumi-las. Vejamos um exemplo.

Abra o ServerModulePrincipal e crie uma function no escopo public nomeando-a como HTTPExemplo conforme mostrado a seguir:

```
function TServerModulePrincipal.
HTTPExample(Teste: String)
 : String;
begin
  Result := Teste;
end;
```

Perceba que é um método muito simples, que nada mais faz do que retornar uma String recebida como parâmetro. A sintaxe é bem simples de ser compreendida conforme mostrado a seguir:

```
http://MeuWebSite:Numero da porta/datasnap/rest/
Nome da
 Classe Server Module/Nome do método/Parâmetros
```

No nosso caso, estamos adotando a porta padrão 80 para a simulação, então execute sua aplicação e faça um teste, digitando no browser o endereço: http://localhost/datasnap/rest/TServerModulePrincipal/HTTPExample/Seu Nome. Veja o resultado do teste na **Figura 8**.

Além disto, somos capazes de criar um Tunneling de dados interceptando os bytes e aplicando várias técnicas de auxílio a problemas em nosso servidor de aplicação. Basicamente, devemos implementar os eventos de interceptação disponíveis no componente DSHTTPService. Esta tarefa é realizada pela manipulação dos seguintes eventos: OnOpenSession, OnErrorOpenSession, OnCloseSession, OnWriteSession, OnErrorWriteSession, OnReadSession e OnErrorReadSession. Um exemplo de chamada de um dos eventos é demonstrado a seguir:

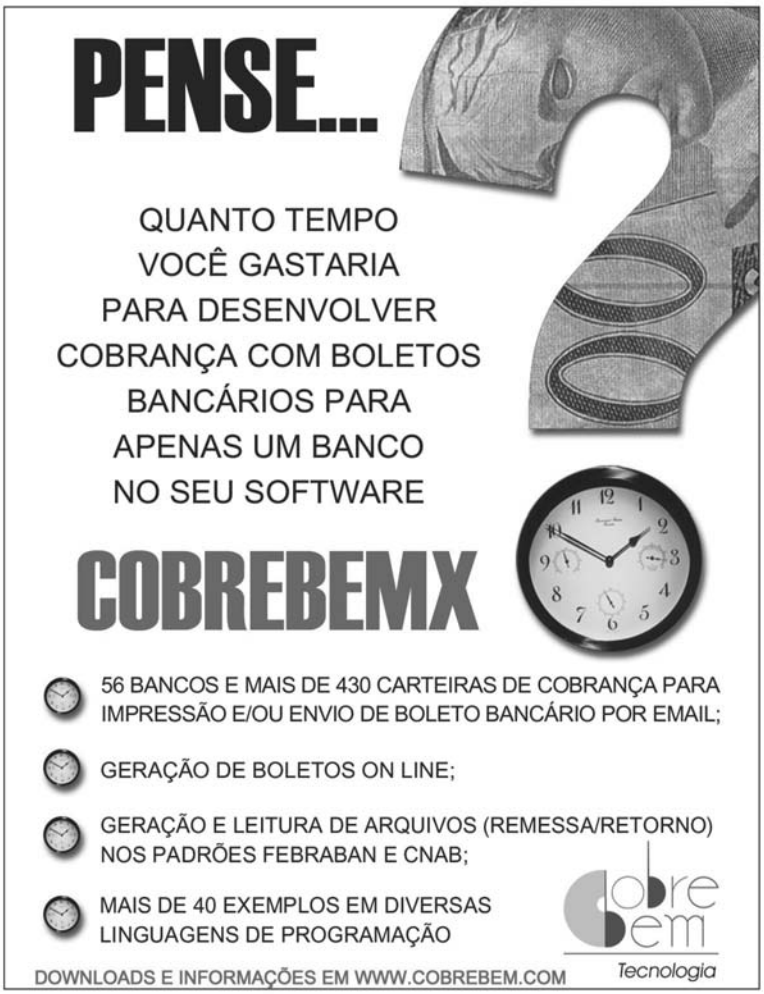
```
DSHTTPService.HttpServer.TunnelService.
OnOpenSession := MeuMetodoAbertura;
```

Outra novidade da nova versão, que é facilmente configurada sem trabalho algum, é a capacidade de filtro de compressão de dados através do ZLibCompression. Selecione o componente DSTCPServerTransport e veja que foi acrescentada a propriedade Filters. Clique sobre TTransportFilterCollection para que seja exibido o editor. Adicione um novo filtro e veja no Object Inspector a propriedade FilterId. Clicando sobre ela você verá que é exibido o ZLibCompression. Realizando estes passos, com pequenos cliques você já possui compressão de dados em seu servidor de aplicação. Os pontos mais interessantes é que esta opção também está disponível no componente DSHTTPService e que esta funcionalidade não necessita de nenhuma configuração adicional à camada cliente, sendo

Figura 8. Exemplo de acesso a métodos do DataSnap através de REST

facilmente registrada e já estando disponível para utilização. É possível criar outros filtros facilmente através de packages e da implementação da classe TTransportFilter. Uma boa dica é olhar a unit DbxCompressionFilter que possui a implementação da compressão de dados nativa na qual foi apresentada.

Após ter finalizado estas modificações e conhecer um pouco das novidades, vamos criar nossa aplicação cliente. Portanto, clique em File>New>VCL Forms Application para dar início à camada cliente. Adicione ao novo formulário um SQLConnection, um DSProviderConnection, um ClientDataSet, um DataSource, um DBGrid e um DBNavigator. Faremos um simples cadastro utilizando a DBGrid para inserção de dados. Lembrando



que esta não é uma boa prática aplicável a um ambiente multicamadas, porém, você pode adaptar o exemplo conforme o que fizemos no caso da apresentação do DBX + Firebird. Renomeie os componentes respectivamente para sqlConCliente, dspcCliente, cdsCliente e dsrCliente. Execute o servidor de aplicação, selecione o SQLConnection e configure seu Driver para DataSnap ativando-o logo em seguida. Selecione o dspcCliente e na propriedade ServerClassName informe o TServerModulePrincipal. No ClientDataSet, aponte o RemoteServer para dspcCliente e indique seu DataSetProvider em ProviderName. Ligue a DBGrid e o DBNavigator ao DataSource. Não se esqueça de adicionar a unit DbxCompressionFilter para possibilitar a utilização dos filtros registrados em nosso servidor de aplicação. Codifique os eventos AfterPost e AfterDelete do ClientDataSet para realizar a gravação no banco de dados conforme o código que segue:

```
cdsCliente.ApplyUpdates(0);
```

Em seguida faremos a abertura do DataSet, no evento OnShow digite:

```
cdsCliente.Close;
cdsCliente.Open;
```

Com isto temos uma aplicação cliente já em funcionamento. Salve seu exemplo nomeando a unit como untPrincipal.pas e o projeto como AppClient.dproj. Conheceremos agora a última novidade que abordarei neste artigo. O uso de Callbacks.

Implementando Callbacks

O conceito de Callback baseia-se em uma chamada da aplicação cliente à servidora, onde a mesma passa respostas para esta chamada de uma forma síncrona. Para quem desenvolvia nas tecnologias anteriores ao DataSnap 2009 talvez já conhecia ou tenha ao menos ouvido falar nesta técnica. Nos modelos anteriores de aplicações multicamadas tínhamos a dependência do COM+, desta forma podíamos criar métodos de chamada via Type Library e realizando as chamadas como um método qualquer, porém como sabemos na versão 2009 a dependência de objetos COM foi removida do novo modelo de desenvolvimento, e este acabou sendo um dos quesitos solicitados por quem adota esta arquitetura.

Na nova versão é possível realizarmos o callback através de comunicação e implementação via JSON. Para isto, devemos implementar uma classe TDBXCallback que provem da nova unit DBXJSON, isto porque JSON nos permite trafegar qualquer tipo de objeto utilizando-se por exemplo um JSONObject. Demonstraremos na prática seu funcionamento de uma forma simples, alterando nosso servidor de aplicação, portanto abra seu projeto AppServer e em seguida o ServerModulePrincipal e crie um procedimento no escopo public conforme ilustra a **Listagem 4.**

Repare que temos dois parâmetros de entrada, sendo o primeiro uma String de entrada e o segundo é derivado da classe TDBXCallBack. Declaramos uma variável do tipo JSONObject que nos permite manipular informações de qualquer tipo. Em seguida utilizamos o método AddPair para adicionar um par de informações que no caso, devolverá a String recebida da aplicação cliente concatenada à mensagem do servidor. Por último a classe Callback que é recebida como parâmetro, executa o método Execute passando como parâmetro nosso JSONObject(CallBackServer). Com isto, finalizamos a criação do método de CallBack no servidor de aplicação. Faremos agora a chamada do método na aplicação cliente.

Primeiramente execute o servidor de aplicações, abra o projeto cliente, clique com o botão direito sobre o sqlConCliente e

Nota do DevMan

A técnica de Tunneling, como o próprio nome sugere, é uma forma de criar um túnel de dados, permitindo a interceptação do byte que é enviado/trafegado pela rede. Desta forma, podemos criar inúmeras soluções que nos permitam realizar a manipulação destes dados utilizando as técnicas de distribuição ou balanceamento de cargas entre servidores, criar métodos de replicação de dados ou mesmo técnicas de FailOver.

Nota do DevMan

FailOver é outra técnica que foi implementada na nova versão, e pode ser facilmente manipulada através dos novos componentes DSHTTPService e DSHTTPServiceAuthenticationManager. Consiste em tornar transparente a ativação de outros servidores de aplicação, caso o servidor principal saia fora do ar ou sofra algum erro grave que tire-o do ar.

Listagem 4. CallBack

```
procedure TServerModulePrincipal.CallBackExample(Msg: String;
  CallBack: TDBXCallBack);
var
  CallBackServer : TJSONObject;
begin
// Adicione DBXJSON ao uses
  CallBackServer := TJSONObject.Create;
  CallBackServer.AddPair(TJSONPair.Create
   ('Mensagem do servidor : ', Msg));
  CallBack.Execute(CallBackServer);
end;
```

Listagem 5. Classe TCallBackClient

```
01 private
02   type
03     TCallBackClient = class(TDBXCallBack)
04     private
05       FCallBackMethod : TCallBackMethod;
06     public
07       constructor Create(CallBackMethod : TCallBackMethod);
08       function Execute(const Arg: TJSONValue): TJSONValue;
          override;
09     end;
10     { Private declarations }
11 public
12   FMyCallBack : TCallBackClient;
13     { Public declarations }
14 end;
```

selecione a opção Generate DataSnap client classes. Com isso, será gerada a unit de métodos do servidor que nos possibilita a utilização na AppCliente. Salve-a com o nome de untProxy.pas e em seguida adicione-a ao Uses do formulário principal juntamente com a unit DBXJSON.

Conforme dito anteriormente, para utilizarmos o CallBack é necessário implementar a classe TDBXCallBack. Antes de mais nada vamos criar abaixo da palavra reservada type um método anônimo, que será explicado a seguir. Faça como o exemplo:

```
type
  TCallBackMethod = reference to function
   (const Args: TJSONValue): TJSONValue;

  TfrmCliente = class(TForm)
...
```

Perceba que declaramos um método chamado TCallBackMethod que referencia uma função que recebe e retorna também um JSONValue. Criado nosso método, faremos a classe conforme visto na **Listagem 5**.

A construção da classe na verdade é muito simples como podemos observar. Basicamente utilizamos o método construtor que pode ser definido como você desejar. No nosso caso foi definida a referência ao método anônimo anteriormente criado. Isso significa que ao criar a classe, obrigatoriamente teremos que passar como parâmetro um método anônimo. O método Execute foi sobrecarregado como pudemos perceber e possui declaração idêntica ao método anônimo. No escopo private declaramos uma variável do mesmo tipo do método anônimo, e na sessão public (do formulário), foi declarada uma variável referente à nossa classe de CallBack. Veja a seguir a implementação do construtor da classe que apenas atribui o parâmetro de entrada à variável anteriormente declarada:

```
constructor TfrmCliente.TCallBackClient.
Create(CallBackMethod: TCallBackMethod);
begin
  FCallBackMethod := CallBackMethod;
end;
```

O método Execute simplesmente realiza a chamada ao método anônimo. Como o método anônimo retorna um TJSONValue, passamos este retorno em nossa função como você pode visualizar a seguir:

```
function TfrmCliente.TCallBackClient.Execute(const Arg:
 TJSONValue): TJSONValue;
begin
  Result := FCallBackMethod(Arg);
end;
```

Faremos agora a inicialização da variável FMyCallBack, realizando a instanciação do método anônimo no evento OnCreate do formulário. Codifique o evento conforme mostra a **Listagem 6**.

Analisando a **Listagem 6** temos:

Listagem 6. Evento OnCreate do formulário padrão

```
01 procedure TfrmCliente.FormCreate(Sender: TObject);
02 begin
03   FMyCallBack := TCallBackClient.Create(
04       function (const Args : TJSONValue) : TJSONValue
05       var
06         jObj : TJSONObject;
07         i    : Integer;
08       begin
09         jObj := TJSONObject(Args);
10         for i := 0 to jObj.Size - 1 do
11           with jObj.Get(i) do
12           begin
13             ShowMessage(Format('%s "%s"', [JSONString.Value,
                JSONValue.Value]));
14             Result := TJSONTrue.Create;
15           end;
16       end);
17 end;
```

• Linhas 03 e 04: Inicialização da variável FMyCallBack criando o método anônimo;
• Linhas 05 a 07: Declaração das variáveis;
• Linhas 09: Criação do nosso JSONObject recebendo os argumentos como parâmetros;
• Linhas 10 a 13: Realizamos a contagem dos argumentos recebidos do servidor exibindo-os através de um ShowMessage;
• Linha 14: Como nosso método anônimo aguarda um retorno (TJSONValue) , retornamos um argumento qualquer, no caso um TJSONTrue;

Adicione ao seu projeto um Button e em seu evento OnClick faça a chamada do método do servidor através da classe Proxy. Veja o exemplo na **Listagem 7.**

Listagem 7. OnClick do botão

```
var
  proxy : TServerModulePrincipalClient;
begin
  sqlConCliente.Open;
  proxy := TServerModulePrincipalClient.Create(
    sqlConCliente.DBXConnection);
  proxy.CallBackExample('Callback Daniel', FMyCallBack);
```

Repare que simplesmente declaramos uma variável Proxy do tipo da classe que estamos evocando. Em seguida realizamos a abertura da conexão passando seu DBXConnection para o método. Na chamada do CallBackExample enviamos um texto qualquer e passamos como parâmetro o método FMyCallBack anteriormente declarado. Repare que com isto, o método execute da AppClient será executado através do servidor. Veja o resultado final do nosso CallBack na **Figura 9**.

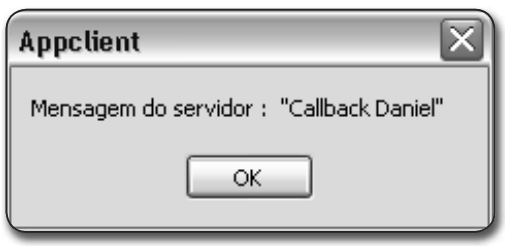


Figura 9. Mensagem retornada do servidor através do CallBack da aplicação cliente

Conclusão

Como vimos no artigo, esta sem dúvida tem tudo para ser a melhor versão do Delphi lançada até hoje. O que vimos por aqui é apenas um breve exemplo de todo o poder da nova ferramenta da Embarcadero. Vimos todo o processo de integração, novidades dos Frameworks DataSnap e DBX e conhecemos alguns dos novos e poderosos recursos da ferramenta. Não deixem de acompanhar os próximos artigos da ClubeDelphi, pois teremos muito mais novidades, exemplos complexos e muitas outras coisas a vermos juntos. Mais uma vez espero que tenham gostado do artigo. Um grande abraço e até a próxima! ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Links

Novidades do RAD Studio 2010
http://www.embarcadero.com/products/rad-studio/whats-new

Download do Firebird 2.0
http://sourceforge.net/projects/firebird/files/firebird-win32/2.0.5-Release/Firebird-2.0.5.13206_1_win32.exe/download

Download do IBExpert Personal
http://ibexpert.net/ibe/index.php?n=Main.DownloadFree

Twitter do Autor
http://www.twitter.com/daniel_laporte

DevSpace do Autor
http://www.devmedia.com.br/daniel_laporte

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Bugs e soluções

Identificação, acompanhamento, correção e prevenção

Do que trata o artigo

Neste artigo daremos uma atenção especial aos bugs, sua identificação, correção e prevenção. Veremos na prática como solucionar alguns dos bugs mais comuns na vida do desenvolvedor Delphi, além de entender porque eles ocorrem. Mostraremos ainda como utilizar uma ferramenta de BugTracking e Profiling para aumentar a qualidade de seus projetos.

Para que serve

Desenvolver projetos com mais qualidade e estabilidade, dando dicas de como atuar de forma preventiva para evitar que erros comuns ocorram e controlando todo o ciclo de vida dos bugs, além de introduzir o leitor o uso de ferramentas de Profiling para melhora de performance e identificação de memory leaks.

Em que situação o tema é útil

Sempre que criarmos interfaces similares, porém sem grau de parentesco entre si, mas mesmo assim relacionadas. Por exemplo, quando desejamos encapsular a implementação de uma interface A em uma classe B que já implementa a interface B, sem precisar implementar os métodos da Interface A explicitamente, muito menos precisar tornar esta classe B um wrapper de outra classe A qualquer que implemente a interface A, economizando tempo e evitando códigos que misturem conceitos e responsabilidades.

Ricardo Coelho
rcoelhorj@gmail.com
Possui graduação em TI, desenvolve software em delphi há cinco anos, estando envolvido principalmente em projetos ligados à telecomunicações. Hoje é coordenador de desenvolvimento na Sinapse Informática. É também responsável pela definição, análise e melhoria dos processos de desenvolvimento da empresa.

Resumo do DevMan

Este artigo abordará um tema que, infelizmente, está presente na vida de todo desenvolvedor, os bugs. Aqui veremos como identificar, corrigir e prevenir alguns dos erros mais comuns do Delphi, além de mostrarmos como utilizar uma ferramenta de Bug Tracking para controlar as solicitações de mudanças e o registro dos bugs encontrados no sistema. Veremos também como utilizar uma ferramenta de Profiling para auxiliar na detecção de memory leaks e gargalos de performance. Veremos tecnologias e conceitos a respeito de: Delphi, BugZilla, BugTracking, Qualidade de software e Profiling.

Todo desenvolvedor, principalmente os que cuidam da manutenção de sistemas, lidam diariamente com diversos bugs, sendo que alguns deles são bem corriqueiros, como erros de conversão, violação de chaves, Access Violations, dentre outros. O que ocorre é que, em alguns casos, o problema parece nunca acabar. Você corrige um bug e na semana seguinte eis que surge de novo em outro lugar, um mês depois aquele que você corrigiu volta a ocorrer, enfim, parece que os bugs não acabam. Por falar nisso, é comum ouvirmos falar que não existem sistemas sem falhas, sem erros. De fato, hoje com minha experiência posso dizer que essa afirmação é verdadeira, apesar de desejar e muito um dia provar que a mesma é falsa. Mas o que ocorre em grande parte dos casos é que um determinado erro ocorre, o desenvolvedor corrige e libera para o cliente, mas não se atenta à origem do problema, perdendo assim uma oportunidade de aumentar a qualidade do sistema inserindo nele rotinas preventivas, para que erros ocorram com menos frequência.

Além de não tratar a causa do problema (possibilitando que o erro volte a ocorrer), normalmente não há um controle do ciclo de vida de um bug, ou seja, da fase de detecção, análise, desenvolvimento da solução e fechamento do mesmo. Com isso, perdem-se prazos e problemas pequenos se tornam grandes dores de cabeça, além de não se formar uma base histórica de problemas e soluções que possa ser analisada para se buscar melhorias no processo e nos padrões.

Uma outra prática bastante negligenciada é a de testes automatizados. É claro que os mesmos demandam um tempo para que sejam efetuados de maneira eficiente, além de exigirem um certo investimento em ferramentas e treinamentos. Porém, mais óbvio ainda é o fato do aumento de qualidade que tal prática traz para os projetos de software.

Aqui veremos como solucionar alguns dos principais bugs que ocorrem no desenvolvimento de aplicações Delphi. A API de exceções do Delphi é gigantesca, é simplesmente impossível examinarmos todas, o foco não será abordar todas as possibilidades que um erro pode ocorrer, mas sim demonstrar as principais ou pelo menos as mais comuns, assim como a adoção de medidas simples para prevenção e detecção dos mesmos. O mais importante aqui é que você perceba o quão importante é você analisar a origem do problema e possíveis formas de preveni-lo. O que quero é que, após ler este artigo, você se faça as seguintes perguntas sempre que se deparar com um erro e for solucioná-lo:

- Porque aconteceu este erro?
- Como ele poderia ter sido evitado?
- Como posso corrigi-lo?
- Que medidas preventivas posso tomar para evitar que o erro volte a ocorrer?

Nossos exemplos serão bem simples, apenas para fins didáticos. Começaremos então vendo algumas das exceções mais comuns ocorridas em sistemas Delphi. A seguir, estudaremos um mecanismo de Bug Tracking e finalmente uma ferramenta para Profiling.

Could not convert variant of type (...) into type (...)

Descrição do erro: Ocorre quando o Delphi tenta atribuir uma variável do tipo variant a outra de outro tipo que não suporta o valor atual da primeira.

Quando ocorre: Este erro é bem comum quando trabalhamos com o componente DBLookupComboBox, pois sua propriedade KeyValue (que retorna o valor do campo definido em KeyField) é do tipo Variant. Dessa forma, se tentamos usar o KeyValue para, por exemplo, exibi-lo em uma mensagem ou concatenar com uma String, recebemos uma exceção.

Na **Figura 1** temos um formulário com um DBLookupComboBox ligado a um DataSource que por sua vez está ligado a um ClientDataSet. A **Listagem 1** mostra o código do OnClick do botão.

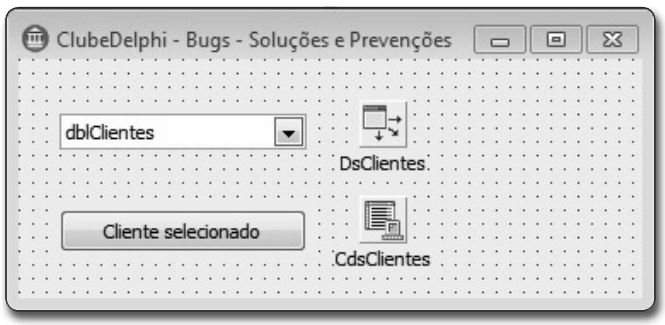


Figura 1. Formulário de exemplo

Listagem 1. Acessando o KeyValue

```
procedure TFrmPrincipal.BtnClienteSelecionadoClick
  (Sender: TObject);
begin
  //Exibe o valor do campo chave
  ShowMessage(dblClientes.KeyValue);
end;
```

Conforme podemos ver no código, ao clicar no botão Cliente selecionado vamos exibir o valor do campo chave do DBLookupComboBox. Caso o usuário selecione um cliente no lookup não teremos problema algum, pois KeyValue estará preenchido, o problema é quando o usuário não seleciona nenhum cliente e clica no botão. Como o valor default é Null, teremos a seguinte exceção exibida na **Figura 2**.

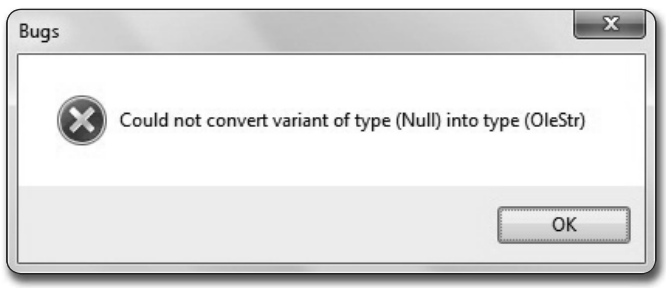


Figura 2. Exceção gerada

A própria mensagem já lhe diz qual o tipo esperado na conversão (OleStr). Com base nisso, basta ver no bloco de código onde está sendo passada uma Variant para uma String para identificar o problema.

Como solucionar e prevenir: Para solucionar este problema poderíamos usar um IF para verificar se existe um valor selecionado no DBLookupComboBox ou então usarmos a função VarToStrDef, conforme mostrado na **Listagem 2**. O resultado final será o mesmo tanto com uso do IF quanto com o uso do VarToStrDef, ambos solucionam o problema e evitam que qualquer erro ocorra neste trecho de código.

Listagem 2. Tratamento do erro de conversão

```
procedure TFrmPrincipal.BtnClienteSelecionadoClick
 (Sender: TObject);
begin
  //Se não for null, exibe, caso contrário exibe um valor
    default.
  if not(dblClientes.KeyValue = null) then
    ShowMessage(dblClientes.KeyValue)
  else
    ShowMessage('Nenhum cliente selecionado.');
end;

procedure TFrmPrincipal.BtnClienteSelecionadoClick
 (Sender: TObject);
begin
  //A função VarToStrDef verifica se a variant é null e se for,
   retorna o
  //valor passado no segundo parâmetro
  ShowMessage(VarToStrDef(dblClientes.KeyValue,'Nenhum cliente
   selecionado.' ));
end;
```

Key Violation

Descrição do erro: Significa que não foi possível prosseguir com a operação, pois geraria uma duplicidade de registro chave no banco de dados ou em um ClientDataSet na memória.

Quando ocorre: Este erro é bem comum quando não trabalhamos com ID's únicos de objetos (campos autoincrementos / generators / sequence), prática normalmente vista em sistemas legados. Tais sistemas normalmente possuem chaves lógicas, ou seja, as chaves primárias do banco de dados são as chaves que o usuário conhece e manipula no seu dia-a-dia, como por exemplo CPF, código, dentre outras.

Na **Figura 3** temos um formulário com um DBGrid e os componentes de acesso a dados (Query + DataSetProvider + ClientDataSet). Neste exemplo estamos com uma tabela de clientes carregada no ClientDataSet, esta tabela tem como chave primária o campo COD_CLIENTE. Implemente os códigos antes de executar sua aplicação conforme a **Listagem 3.**

Listagem 3. Eventos implementados no ClientDataSet

```
procedure TForm1.CdsClienteAfterPost(DataSet: TDataSet);
begin
  //Execução do applyUpdates para gravar em banco as alterações
    feitas no
  //ClientDataSet toda vez que for dado um post no mesmo
  CdsCLiente.ApplyUpdates(0);
end;

procedure TForm1.CdsClienteReconcileError
 (DataSet: TCustomClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind; var Action:
   TReconcileAction);
begin
  //Implementação do evento ReconcileError para capturarmos erro gerados
  //pelo ApplyUpdates.
  ShowMessage(E.Message);
  Action := TReconcileAction.raCancel;
end;
```

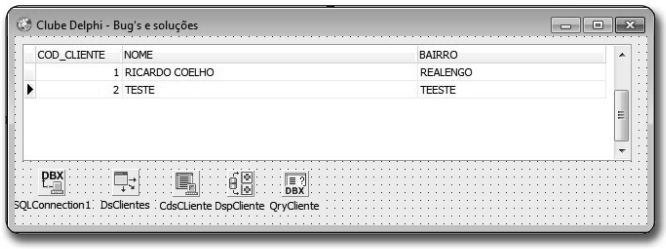


Figura 3. Formulário com componentes

Ao tentarmos gravar um novo cliente com o código 2, temos o erro gerado pelo evento OnReconcileError conforme ilustra a **Figura 4**.

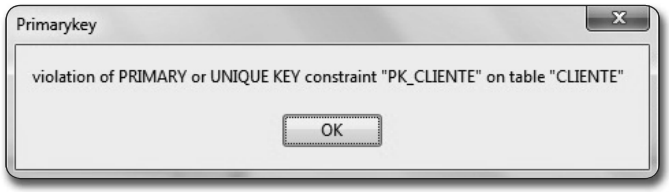


Figura 4. Erro gerado ao tentar gravar cliente com código duplicado

Este é um erro clássico de violação de chave primária no banco de dados, porém, ele só ocorreu porque nosso ProviderFlag não está configurado corretamente, sendo assim, vamos configurar o ProviderFlag do campo COD_CLIENTE e tentar executar de novo a aplicação (**Figura 5**).

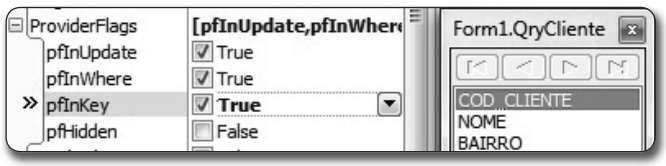


Figura 5. Configuração do ProviderFlag do campo COD_CLIENTE

Ao tentarmos incluir um novo cliente com o código 2, agora obtemos a exceção conforme mostrado na **Figura 6**.

Figura 6. Exceção gerada pelo ClientDataSet

Como podemos ver, a mensagem gerada após configurarmos o ProviderFlag foi diferente da anterior, isto ocorre porque esta mensagem é gerada pelo ClientDataSet ao realizar o Post, devido à configuração do ProviderFlag onde estabelecemos que COD_CLIENTE é a nossa chave primária. A grande diferença entre a primeira mensagem e a segunda é que a primeira foi gerada quando o SQLQuery tentou executar a operação no banco de dados, enquanto que a segunda foi gerada quando o ClientDataSet tentou dar um post, baseada na configuração

do ProviderFlags, ou seja, neste segundo exemplo o DataSetProvider nem chegou a ser acionado. Sabendo-se disso fica mais fácil identificar o erro, pois pela própria mensagem já conseguimos saber se a exceção foi gerada pelo ClientDataSet ou pela Query.

Como solucionar e prevenir: poderíamos criar uma rotina no BeforePost para verificar se o registro que está sendo inserido já está no banco e exibirmos uma mensagem amigável ao usuário, mas isso pode se tornar trabalhoso para se fazer em cada DataSet, sem contar que antes de efetivar a gravação teríamos que fazer uma consulta no banco de dados. Uma alternativa é realizar o tratamento nos próprios eventos de erro do ClientDataSet. Codifique os eventos do ClientDatSet conforme a **Listagem 4** e execute novamente sua aplicação para perceber que os dois casos exibidos foram tratados corretamente.

Listagem 4. Novos eventos implementados no ClientDataSet

```
procedure TForm1.CdsClientePostError(DataSet: TDataSet;
  E: EDatabaseError;
  var Action: TDataAction);
begin
  //Verifica se a mensagem é KEY VIOLATION e exibe mensagem
   amigável
  if (pos('KEY VIOLATION',UpperCase(E.Message))) > 0 then
    ShowMessage('Existe um registro com a mesma chave.');
  //Aborta a operação
  Action := daAbort;
end;

procedure TForm1.CdsClienteReconcileError(DataSet:
TCustomClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind; var Action:
   TReconcileAction);
begin
  //Verifica se a mensagem é PRIMARY KEY e exibe mensagem amigável
  if (pos('PRIMARY OR UNIQUE KEY',UpperCase(E.Message))) > 0 then
    ShowMessage('Existe um registro com a mesma chave.');
  //Cancela a operação
  Action := TReconcileAction.raCancel;
end;
```

Record not found or changed by another user

Descrição do erro: indica que o DataSetProvider não conseguiu encontrar o registro especificado para realização de um update ou delete.

Quando ocorre: devido à alteração do registro em uma outra sessão que impede que o DataSetProvider execute a atualização/exclusão do registro em questão.

Vamos utilizar o exemplo anterior, alterando apenas o ProviderFlag do Field COD_CLIENTE informando False para pfInKey. Agora faça o seguinte teste: abra várias instâncias da aplicação e altere simultaneamente campos do mesmo registro (**Figura 7**).

Nota

O termo sessão aqui indica uma outra instância da aplicação, que pode estar rodando na mesma máquina ou em outra estação da rede conectada ao mesmo servidor de banco de dados.

Como podemos observar, a exceção foi gerada ao tentarmos editar o registro na segunda sessão quando esta estava desatualizada. Isso ocorreu porque o registro foi alterado na primeira sessão e o DataSetProvider não conseguiu encontrá-lo para realizar o update no mesmo, pois como o modo de atualização definido na propriedade UpdateMode foi marcado como upWhereAll, a instrução update foi gerada passando todos os campos configurados como pfInWhere no ProviderFlag, sendo assim, o DataSetProvider gerou uma instrução SQL semelhante a esta:

```
UPDATE CLIENTE SET BAIRRO = 'RECREIO' WHERE COD_CLIENTE =
 3 AND NOME = 'GABRIELLE XAVES' AND BAIRRO = 'BARRA'
```

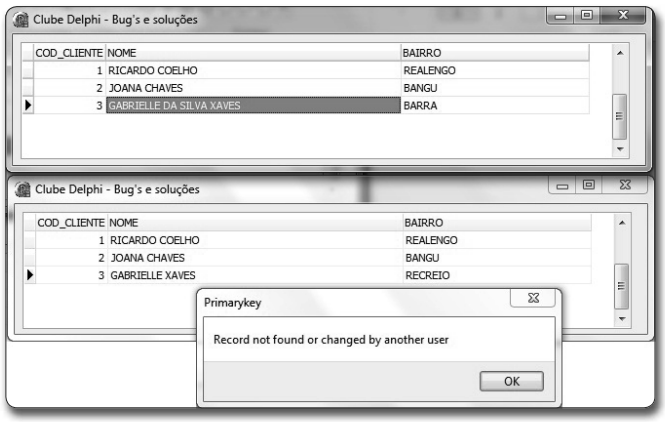


Figura 7. Alteração de registro na segunda sessão

Esta exceção se deve ao fato de que não existe mais nenhum registro com esta condição (devido à alteração alteramos na primeira sessão).

Como solucionar e prevenir: a primeira coisa a fazermos é alterarmos o DataSetProvider para gerar as instruções de Delete e Update baseando-se apenas na chave primária. Com isto, já reduziremos consideravelmente as chances deste problema ocorrer. Para isso, basta alterar a propriedade UpdateMode do DataSetProvider para upWhereKeyOnly, dessa forma, o DataSetProvider irá gerar as instruções SQL baseando-se nos campos definidos como pfInKey nos ProviderFlags. Ao executarmos novamente a aplicação com esta propriedade modificada e tentarmos alterar um registro, teremos o erro "Unable to find record. No key specified". Este erro ocorreu devido ao fato do ProviderFlag estar setado como upWhereKeyOnly, porém, não definimos nenhum campo chave nos ProviderFlags. Para solucionarmos, basta alterarmos o ProviderFlag do campo COD_CLIENTE, marcando a opção pfInKey como True.

Com esta configuração não teremos mais o erro apresentado, além de resolver o problema da edição simultânea apresentada anteriormente, isto para os casos em que o campo alterado não é a chave primária. Com a nova configuração a instrução SQL gerada pelo DataSetProvider será semelhante à seguir:

```
UPDATE CLIENTE SET BAIRRO = 'RECREIO' WHERE COD_CLIENTE = 3;
```

Nota

Note, obviamente, que neste caso o usuário que fizer o post por último é o que terá o registro devidamente atualizado.

Agora que conhecemos alguns dos erros mais comuns do Delphi e como solucioná-los, vamos examinar uma ferramenta onde podemos cadastrar os bugs encontrados na aplicação, para que possam ser gerenciados.

Bug Tracking

Um Bug Tracking nada mais é do que uma prática para auxiliar na qualidade de um software, auxiliando no registro e controle da evolução dos bugs. Boa parte das ferramentas desse tipo é desenvolvida para a plataforma Web e permite que o próprio usuário abra um chamado para correção do bug diretamente. Isso é interessante tanto para o cliente quanto para a empresa desenvolvedora do software, pois para o cliente é permitido o lançamento e acompanhamento de toda a evolução de seu chamado, enquanto que para a empresa fornecedora do software é possível a centralização das solicitações de mudança, sem contar na padronização da abertura de chamados. Dentre as ferramentas disponíveis podemos citar o BugZilla, o Mantis e o Trac, todas são open-source e fornecem uma interface Web para lançamento e acompanhamento dos bugs. Neste artigo nós falaremos do BugZilla.

Os bugs lançados no BugZilla são baseados em produtos e componentes. Os produtos são os sistemas de sua empresa, enquanto que os componentes são as partes dele. Essa estrutura básica facilita o registro dos bugs por parte dos usuários e também na medição das partes mais problemáticas da aplicação, indicando talvez uma necessidade de refatoração da mesma. O BugZilla permite ainda o lançamento de informações que auxiliam o desenvolvedor na abertura de um chamado, como por exemplo, hardware e sistema operacional. Além disso, é possível definir a severidade e prioridade de um bug, ajudando a equipe de desenvolvimento na priorização das correções.

Conhecendo o BugZilla

Após instalar e configurar o BugZilla, temos a página inicial do aplicativo, onde vemos três opções:

- File a Bug: Permite reportar um novo bug;
- Search: possibilita buscar e verificar os bugs já registrados no sistema e seus respectivos status;
- Open a New Acount: realiza o cadastro de uma nova conta de usuário, nesta opção será solicitado que o usuário informe seu email que será também o seu login dentro do BugZilla. Ao solicitar a criação de uma nova conta, o BugZilla enviará um email de confirmação para o email informado no cadastro.

Nota

Não abordaremos aqui a instalação do BugZilla e nem mesmo todas as suas opções. O foco será a customização e uso básico do Bug Tracking, demonstrando como ele pode nos auxiliar na rastreabilidade de problemas, assim como na melhoria da qualidade de produtos de software através do acompanhamento das soluções de bugs e da medição das partes mais problemáticas do sistema. Para download e instalação do BugZilla consulte a sessão Links ao final do artigo.

Na parte superior, podemos visualizar os campos de login e senha. Nós iremos utilizar a conta de administrador (informada durante a instalação) para conhecermos e realizarmos algumas configurações básicas, porém suficientes, para termos um controle eficiente.

Customizando o BugZilla

Após logarmos com a conta de administrador e acessarmos o menu Administration, veremos uma página com diversas opções de customização e tarefas administrativas.

A primeira configuração essencial está dentro da sessão Parameters. Após acessar essa opção, podemos ver ao lado esquerdo as opções de configuração, e no corpo da página os campos que podemos configurar. Inicialmente vemos a opção Required Settings, nela o que nos interessa é o campo maintainer, pois, nele definimos o email do mantenedor do BugZilla que seria o administrador da ferramenta. Este email será usado pelo BugZilla para exibir para os usuários a ocorrência de algum erro no sistema. Na opção Bug Fields, vamos habilitar a opção use classification, que nos permitirá criar classificações para os bugs.

Criando e gerenciando usuários

Ao acessarmos a opção users no painel de administração veremos uma página onde há a possibilidade de pesquisarmos as contas existentes ou de criarmos um novo usuário (**Figura 8**). Note na tela, que após a criação do usuário, podemos especificar diversas opções, como os grupos de acesso. Note que marcamos a opção Bugmail Disabled, pois, não efetuamos as configurações de SMTP, dessa forma o BugZilla não enviará e-mails de notificação para este usuário.

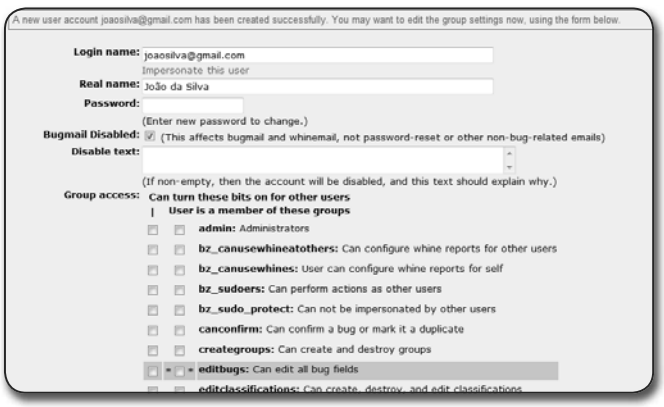


Figura 8. Definição de permissões da conta recém criada

Criando classificações

Voltando ao painel de administração, temos a opção Classifications, que permite a criação de novas classificações para nossos softwares. As classificações são uma forma de agrupar os produtos tratados pelo BugZilla, facilitando as buscas de bugs e também em seus lançamentos. Podemos criar quaisquer classificações para nossos produtos conforme mostrado na **Figura 9**.

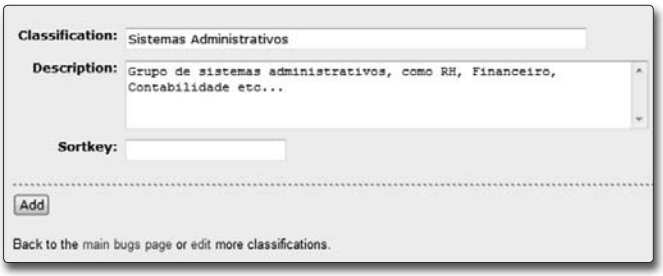


Figura 9. Criação de uma nova classificação

Nota

Em diversas telas pode-se ver o campo SortKey que nada mais é do que um valor qualquer, definido para que o BugZilla realize a ordenação da informação em questão.

Criando produtos

Voltando ao painel de administração, temos a opção Products, que permite a catalogação dos produtos de nossa empresa. Os produtos são os nossos softwares, eles nos permitirão medir, por exemplo, quais sistemas possuem maior índice de bugs reportados. Ao acessar a opção Products, já visualizamos as classificações existentes. Ao clicarmos em Add Product, criaremos um novo produto para a classificação desejada, assim como seus componentes e versões. Ao confirmarmos a criação do produto clicando em Add, visualizaremos a tela exibida na **Figura 10**. Observe que o item Edit components está em destaque. Isso ocorre devido à criação de um registro de produto, porém, sem que haja algum componente cadastrado. Vamos clicar em Edit components para adicionarmos os componentes deste produto. Informarmos o nome do componente, a descrição e outros três campos, que são: os usuários responsáveis pelo componente, o contato para controle de qualidade e uma lista de contatos que serão notificados quando um email for enviado referindo-se a este componente.

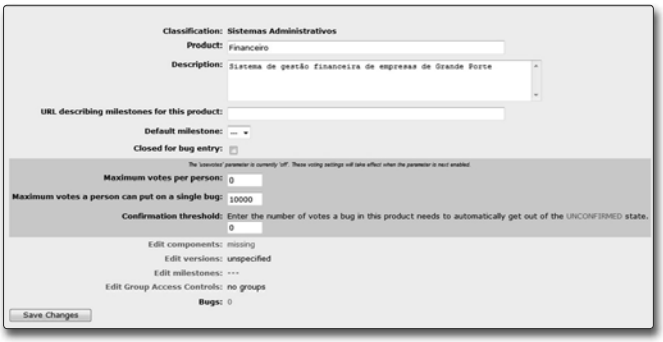


Figura 10. Novo produto criado

Criando campos customizados

O BugZilla permite também criar campos personalizados. Isto é de grande serventia para a customização de um sistema, pois é muito comum usarmos uma ferramenta e a mesma não dispor de informações importantes para o nosso processo interno. Com esta opção podemos criar campos suprindo esta necessidade, no nosso exemplo criaremos o campo Impacto, que servirá para o usuário informar quão impactante é o bug em seu dia-a-dia, nos auxiliando na priorização das soluções. Ao acessarmos a opção Custom Fields no painel de administração, no cadastro de custom fields, veremos uma tela semelhante a da **Figura 11**.

Figura 11. Cadastro de campos customizados

Conforme podemos observar, podemos definir o nome do campo, a descrição e o tipo, ou seja, se será um DropDownList, um campo texto entre outros. Pode-se definir ainda uma condição para exibição deste campo, através da opção Field only appears when, onde poderíamos definir que este campo só apareceria para um determinado produto. No nosso exemplo, determinamos que o campo Impacto será um texto livre e exibiremos este campo sempre que um novo bug for reportado.

Definindo field values

O BugZilla possui alguns campos que são baseados em uma lista de valores pré-definida, como por exemplo sistema operacional, hardware, status do bug dentre outros. Para dar mais flexibilidade, podemos customizar estas listas, deixando-as de acordo com o processo interno definido em nossas empresas. No painel de administração da opção Field Values, podemos alterar estas listas. Iremos incluir um novo status, que se chamará Analising. O resultado da criação do status é exibido na **Figura 12**.

Figura 12. Status recém criado

Definindo o workflow dos status

Um dos recursos mais interessantes do BugZilla é a possibilidade de customização do WorkFlow dos status dos bugs. Cada empresa tem um processo diferente, com status de atendimento diferenciados e este recurso possibilita a customização do

Nota

O BugZilla possui um sistema de votos de bugs. Cada usuário possui um limite total de votos que pode distribuir aos bugs em aberto, determinando assim o grau de relevância de cada um dos bugs.

WorkFlow dos status dos bugs. Por exemplo, em uma determinada empresa, um chamado pode ser aberto e em seguida pode ser diretamente fechado, enquanto que em outra, um chamado só pode ser fechado após passar por uma fase de análise. É um exemplo simples, apenas para ilustrar as diferenças que podem ocorrer em diferentes empresas. No nosso caso, acabamos de criar o novo status Analising, e agora vamos introduzi-lo no nosso processo, informando quais as possíveis transições de estado em que o novo status estará vinculado.

Observando a **Figura 13** podemos notar que um bug só pode ir para análise quando é novo (new) ou quando está reaberto (reopened). Por outro lado, após um bug estar em análise, o mesmo pode ser dado como resolvido, verificado ou fechado.

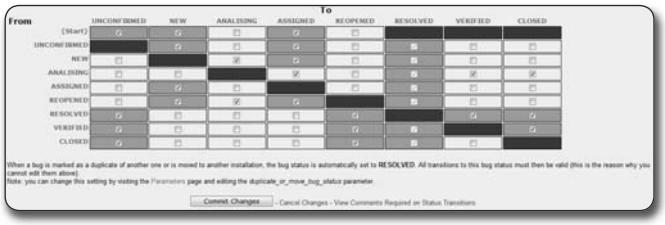


Figura 13. Definição de Workflow

Ao lado do botão Commit Changes temos a opção View Comments Required on Status Transition. Nesta opção, podemos definir quais transições de estado irão exigir que o usuário informe um comentário sobre a transição. Por exemplo, podemos definir que toda vez que um bug tiver seu status alterado de Analising para Verified ou Closed, exigiremos que o usuário informe um comentário, que seria o motivo da transição do estado.

Lançando bugs

Agora que já vimos algumas formas básicas de customização do Bugzilla, vamos ver como o usuário lançaria os bugs reportados. Logo na página principal, ao clicarmos em New, teremos a opção de selecionar o produto no qual desejamos reportar um bug. Após selecionar o produto, teremos a tela de lançamento de bug, como mostra a **Figura 14**. Nesta tela você pode verificar que informamos todas as características do bug, como componente, severidade, prioridade, versão do sistema operacional etc.

Atualizando um bug

Para localizar um bug basta digitar uma palavra chave, clicar em Search e a lista de bugs será exibida. No nosso exemplo, basta digitar a palavra overflow (o que digitamos como sumário) e clicar em search. Para editar um bug, basta clicarmos em seu ID. Assim será aberta a tela para edição do bug. É possível visualizar todas as informações sobre o erro, inclusive quem o reportou, quando foi reportado e um histórico de alterações (**Figura 15**).

Nota

WorkFlow nada mais são do que sequências de tarefas, muito utilizadas em engenharia de software para descrever ou sequenciar processos. Neste caso são utilizadas listas, relatórios ou outras formas que demonstrem evolução por meio de passos.

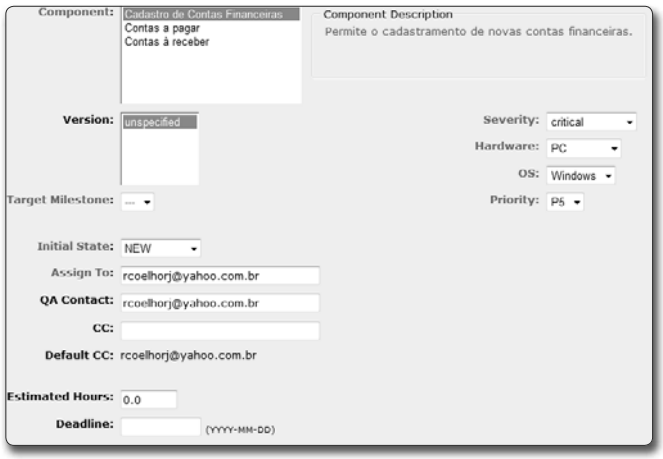


Figura 14. Lançamento de um bug

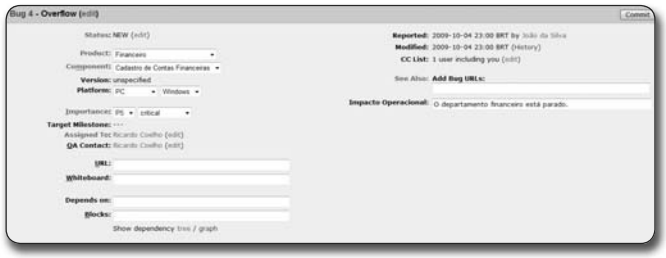


Figura 15. Alteração de bug

Na **Figura 16** podemos ver a possibilidade de lançamento de informações sobre estimativa e prazos, bem como definir um comentário e atualizar o status deste bug (levando em consideração o que definimos no WorkFlow).

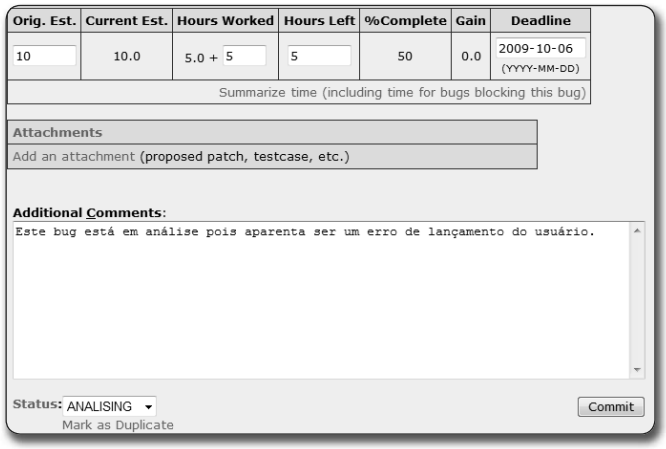


Figura 16. Estimativas e status do bug

É interessante analisarmos que o BugZilla mantém todo o histórico de alterações dos bugs. Se acessarmos novamente este bug e clicarmos na opção histórico, será possível visualizar

todas as alterações realizadas no mesmo além de visualizar também todos os comentários do bug.

Consultando bugs

Ao clicar na opção search da página principal você poderá filtrar os bugs em aberto ou visualizar todos, sem a necessidade de aplicar nenhum filtro, como mostra a **Figura 17**. Ao clicar na opção Long Format, serão exibidos mais detalhes sobre cada bug. Uma outra opção interessante é o Time Summary, que permite realizar uma consulta do tempo gasto, ou por cada desenvolvedor.

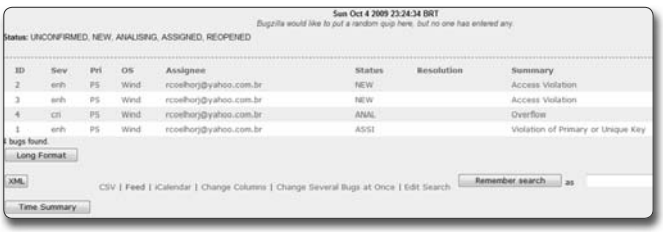


Figura 17. Consulta de bugs

Emitindo relatórios

O BugZilla, além de todos os recursos que vimos até aqui, possui também um gerador de relatórios muito flexível. Ao clicarmos na opção Reports teremos uma série de filtros e opções para emissão de relatórios customizados (**Figura 18**).

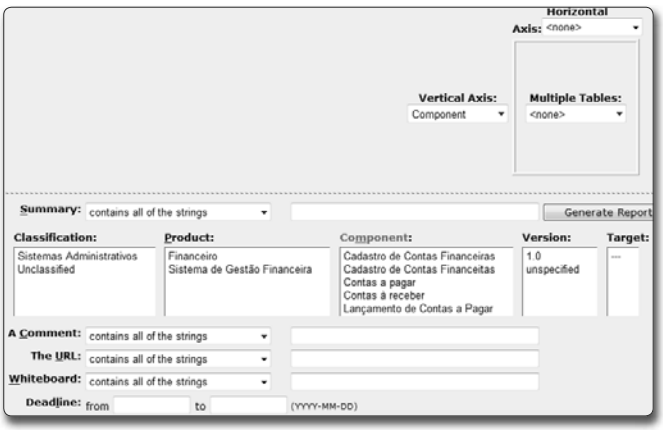


Figura 18. Emissão de relatórios

Observando a **Figura 18**, definimos Component como Vertical Axis, para identificarmos quais os componentes mais apresentam bugs no sistema. Com o resultado do relatório da **Figura 19**, podemos ver que o Cadastro de Contas Financeiras é atualmente o recurso com mais ocorrências, utilizamos apenas opções básicas, mas com isso percebe-se que é possível extrair resultados que realmente nos ajudam na tomada de decisão durante o processo de desenvolvimento e manutenção dos nossos sistemas.

Profiling

Em desenvolvimento de software, Profiling consiste em investigar o comportamento de um software. Através desta prática é possível identificar os "gargalos" do sistema, tomando-se conhecimento do tempo de execução de cada rotina e/ou recursos alocados. Infelizmente, a prática de Profiling de forma preventiva não é muito comum, muito pelo contrário, esta prática é aplicada com maior frequência quando surgem problemas de memory leaks, desempenho e travamentos do sistema. No entanto o ideal seria a realização periódica de tal prática para medir constantemente se o produto gerado não está com desperdícios de recursos, tornando-se uma "bomba-relógio" prestes a explodir a qualquer momento. Como o foco do artigo não é exatamente o uso de ferramentas de Profiling, nós não iremos estender muito o assunto, apesar do mesmo estar relacionado à solução de problemas de software e bugs. A ideia aqui é demonstrar de maneira objetiva, para que o leitor tome conhecimento e compreenda a importância do uso desta prática.

Usaremos em nossos exemplos o AQTime. Trata-se de uma ferramenta completa e que fornece informações suficientes para nos auxiliar na otimização de softwares. Faremos dois exemplos, um para identificação de memory leaks e outro para consumo de memória.

Component		
	Cadastro de Contas Financeitas	2
	Lançamento de Contas a Pagar	1
	Total	3

Figura 19. Resultado do relatório

No nosso primeiro exemplo, vamos simular um vazamento de memória e em seguida veremos como identificá-lo usando o AQTime. Crie uma nova aplicação e no formulário exibido, declare as duas classes como mostra a **Listagem 5.**

Na **Listagem 5** temos as classes TDepartamento e TFuncionario, cada funcionário está vinculado a um departamento, por isto, definimos o departamento no construtor do funcionário, conforme a **Listagem 6**.

No formulário da aplicação insira um Button modificando sua propriedade Name para BtnCriarCliente e dois Edits alterando a mesma propriedade para EdtNome e EdtIdade respectivamente. Em seguida, codifique o evento OnClick como mostrado na **Listagem 7.**

Aqui temos a implementação do evento OnClick do botão para simularmos o memory leak. O código é bem simples, nós criamos um funcionário e alimentamos ele com os valores da tela para em seguida o destruirmos. Nosso exemplo para simular o memory leak está pronto. Agora vamos ao AQTime. Antes de iniciarmos nossas operações no AQTime, precisamos alterar

Nota do DevMan

Memory Leak, também chamado de vazamento de memória, ocorre quando uma porção de memória alocada para um determinado objeto não é liberada, permanecendo ocupada quando não é mais necessária. Por exemplo, quando criamos um objeto, usamos o mesmo e depois quando este não é mais necessário, não o destruímos, ocorrendo assim um vazamento de memória.

Listagem 5. Declaração de classes usadas no exemplo

```
...
type
  TDepartamento = class
  public
    nome : String;
    constructor Create;
    destructor Destroy; override;
  end;

  TFuncionario = class
  public
    Nome : String;
    Idade : Integer;
    Departamento : TDepartamento;
    constructor Create;
    destructor Destroy; override;
  end;
  TForm1 = class(TForm)
...
```

Listagem 6. Inicializando o Departamento

```
constructor TFuncionario.create;
begin
  inherited;
  Departamento := TDepartamento.create;
end;
```

Listagem 7. Criando um funcionário

```
procedure TForm1.BtnCriarClienteClick(Sender: TObject);
var
  f : TFuncionario;
begin
  f := TFuncionario.Create();
  f.Nome  := EdtNome.Text;
  f.Idade := StrToIntDef(EdtIdade.Text,0);
  f.Free();
end;
```

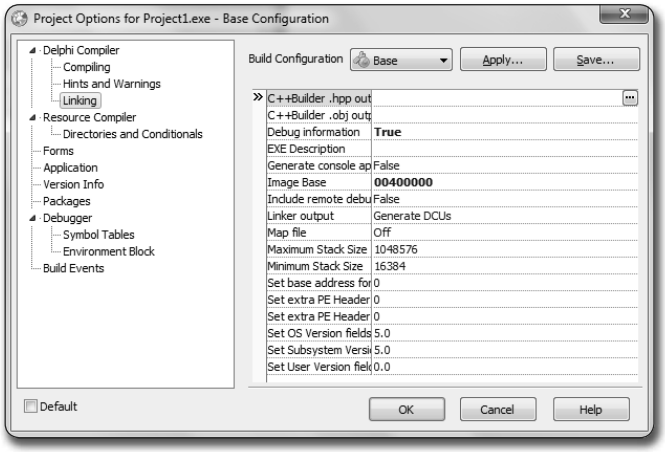


Figura 20. Alteração necessária para uso do exe pelo AQTime

o projeto no Delphi para que nosso executável seja gerado com informações de debug, para permitir que o AQTime faça todo o mapeamento das rotinas de nossa aplicação. Para isso, vá em Project>Options>Linking e marque a opção Debug Information como True, conforme podemos ver na **Figura 20**.

Agora basta gerarmos o executável de nossa aplicação para iniciarmos nossas operações no AQTime. O ambiente do AQTime é dividido em algumas seções básicas (**Figura 21**):

• Setup Modules: São os executáveis do nosso projeto que faremos o Profile;

• Results: Exibe os resultados do Profile da aplicação, com o tempo de execução, consumo de recursos, dentre outras informações;

• Áreas: É onde definimos as partes dos nossos executáveis (modules) que iremos executar o Profiler;

• Allocation Profiler: Onde definimos o tipo de Profiler que executaremos (neste artigo trabalharemos apenas com dois deles).

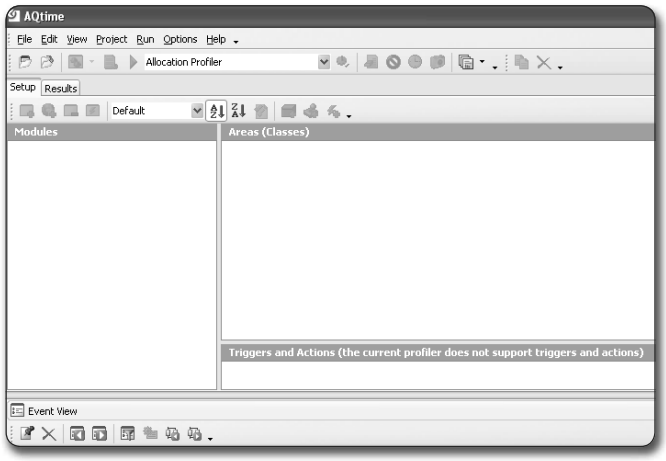


Figura 21. Página inicial do AQTime

Nosso primeiro passo é criar um novo projeto em File>New>Project. Em seguida vamos adicionar nosso executável a este projeto. Para isto, basta clicar com o botão direito na área Modules (opção Add Module). O AQTime carregará todas as informações sobre o nosso executável, inclusive as classes TDepartamento e TFuncionario. Certifique-se de ter selecionado a opção Allocation Profiler no menu superior do AQTime. Agora basta executarmos (seta verde, semelhante à do Delphi) e deixarmos as opções Default marcadas. Após clicarmos em Run, veremos a aplicação ser executada, assim podemos realizar quaisquer operações normalmente, que no nosso caso, digitaremos uma informação qualquer e clicaremos quatro vezes no botão Criar Funcionário, fechando a aplicação em seguida. Veja na **Figura 22** o resultado após o encerramento.

Como podemos ver foram criadas quatro instâncias de TFuncionario e de TDepartamento, sendo que no final da execução do programa não sobrou nenhuma instância de TFuncionario, porém, sobraram quatro instâncias de TDepartamento na memória. Esta informação é exibida na coluna Live Count, que nos diz quantos objetos desta classe permaneceram em memória. Com esta informação em mãos, re-analisando nossa aplicação, podemos perceber que a criação do Departamento ocorre na criação do Funcionário, mas este não é destruído quando ocorre a destruição do Funcionário. Vamos então implementar esta solução e executar novamente o Profiler. Modifique o destrutor da classe TFuncionario conforme a **Listagem 8.**

Listagem 8. Destruindo o departamento

```
destructor TFuncionario.destroy;
begin
  FreeAndNil(Departamento);
  inherited;
end;
```

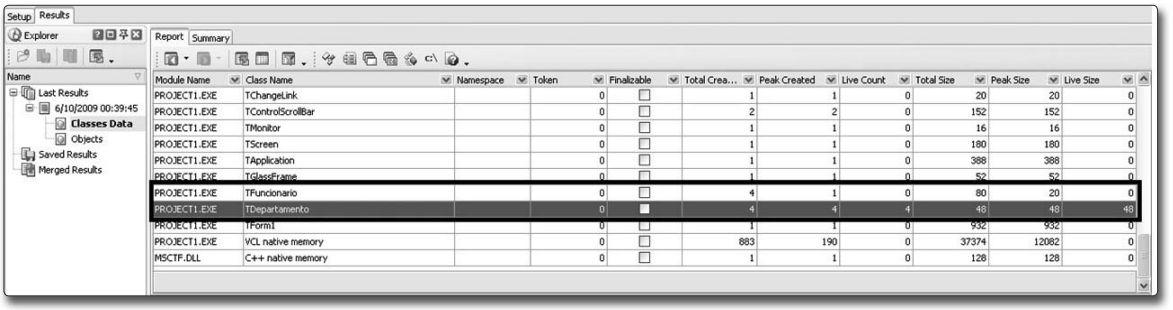


Figura 22. Resultados do Profile

Ao analisarmos o resultado do Profile novamente, percebemos que agora foram criadas quatro instâncias de TFuncionario e TDepartamento, porém nenhuma permaneceu criada, diferentemente da primeira execução quando quatro departamentos permaneceram em memória.

Este mesmo Profiler nos permite também analisar o tamanho dos objetos através da coluna Total Size. Analisando esta coluna podemos avaliar o "peso" de um objeto na memória, por exemplo, imagine que a classe funcionário tivesse 25 campos e que tivéssemos cerca de 5000 funcionários. Imagine que precisássemos carregar estes 5000 funcionários em uma lista para que o usuário pudesse fazer uma busca. Certamente isso consumiria uma quantidade de recursos razoável no sistema.

Conclusão

O objetivo deste artigo não foi somente demonstrar a solução de bugs comuns do Delphi. Vimos uma forma de fazer com que os desenvolvedores, quando estiverem diante de qualquer tipo de bug, se questionem sobre o porquê da ocorrência do erro e principalmente, sobre o que pode ser feito para que o erro não volte a ocorrer, ao invés de simplesmente corrigir sem identificar a possível causa.

Nós vimos também como controlar o ciclo de vida de bugs e solicitações de mudanças através de uma ferramenta de Bug Tracking. Mais importante do que você entender como funciona o BugZilla e suas principais funções, é compreender a importância que o controle de mudanças tem para a manutenção de sistemas e o quão fundamental isso é para uma evolução sustentável de qualquer produto.

Por fim, falamos de outro tema intimamente relacionado à detecção de problemas e bugs na aplicação, a prática de Profiling. Nosso objetivo foi apenas mostrar o quão produtivo pode ser ter uma ferramenta desta em nosso cotidiano, além do quanto pode nos ajudar a desarmar as "bombas-relógio" do desenvolvimento de software. ●

Links

Download do BugZilla

http://www.bugzilla.org/download/#v34

Instalação do BugZilla

https://wiki.mozilla.org/Bugzilla:Win32Install

Download do AQTime

http://www.automatedqa.com/downloads/aqtime

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Existem coisas que não conseguimos ficar sem!

...só pra lembrar, sua assinatura pode estar acabando!

Renove Já!

www.devmedia.com.br/renovacao

DevMedia GROUP

Para mais informações:
www.devmedia.com.br/central

Nesta seção você encontra artigos para iniciantes na linguagem Delphi

Introdução à POO – Parte 5

Aprenda de uma forma simples a programar de forma correta

Do que trata o artigo

Este artigo aborda a orientação a objetos com o Delphi, usando uma metodologia simples, didática, de fácil aprendizado. Veremos na teoria, e também na prática, todos os conceitos, fundamentos e recursos oferecidos pelo Delphi para promover a POO.

Para que serve

A POO pode e deve ser aplicada de forma inteligente, ela serve para construir sistemas mais robustos, mais confiáveis, de fácil manutenção, que permitam maior reaproveitamento de código.

Em que situação o tema é útil

A POO é útil em qualquer sistema, seja ele Web, Desktop, Mobile, não importa o tipo de aplicação. Os conceitos aqui apresentados, em um exemplo simples, podem ser utilizados em aplicações reais, como é apresentado em um estudo de caso no final desta série.

Guinther Pauli
guinther.pauli@gmail.com
Blog: http://guintherpauli.blogspot.com
http://twitter.com/guintherpauli
Tem 22 anos de experiência na área de desenvolvimento e arquitetura de software. Certificado em Delphi 3, 5, 6, 7, 2005, 2006, Delphi para Web e Kylix, certificado Microsoft MCP, MCAD, MCSD.NET, MCTS e MCPD (C#, ASP.NET). Editor da Revista ClubeDelphi e da Revista .NET Magazine Brasil, professor de Pós-Graduação da UNIVEM-SP. Especialista em desenvolvimento Delphi, C#, .NET, ASP.NET, atuando também em projetos de migração orientando empresas e desenvolvedores na adotação de melhores práticas. Foi arquiteto e um dos desenvolvedores, voluntário, do site de doações on-line do projeto Criança Esperança da Rede Globo / Unesco.

Resumo do DevMan

Pacotes são importantes para organizar a forma como nossos componentes são distribuídos. Neste artigo, veremos como criar pacotes e como trabalhar com as sessões contains, requires e dependências. Vamos também examinar como usar UML com Delphi, para modelar nosso pequeno framework. Veremos como criar mais uma classe no framework: TBike, que por tabela já poderá ser usada por TPessoas, graças a abstração. E finalmente, vamos examinar um exemplo que usa interfaces.

No artigo da edição anterior, vimos como dar um grande passo além das classes. Vimos como usar componentes. Praticamente tudo o que fazemos hoje no Delphi é usando, ligando e configurando componentes. Vimos a importância de TPersistent, o uso de RTTI (Runtime Type Information) e serialização de objetos, e como as classes "core" da hierarquia da

VCL do Delphi estão organizadas, como TObject, TPersistent, TComponent, TControl, TGraphicControl e TWinControl. Além disso, vimos como criar pacotes, que ajudam a organizar componentes e units e também ajudam na modularização, seja em design time ou runtime. O mais interessante foi quando usamos abstração e polimorfismo através de componentes, quando uma propriedade Transporte no componente Pessoa abstraiu os vários tipos complexos de meio de transporte (reforçando, isso é programar para interfaces, ou mais exatamente neste caso, uma classe abstrata).

Este será o penúltimo artigo da série sobre orientação a objetos com o Delphi. Vamos estudar mais sobre pacotes (packages) e as importantes sessões contains e requires. Além disso, vamos criar um novo pacote exclusivo para abrigar as classes de meio de transporte e separar da classe que opera os transportes (pessoa). Veremos também o suporte à modelagem UML no Delphi. Vamos ainda estender nosso framework criando mais um meio de transporte e assim comprovar que ele é expansível e plugável, reforçando todos os princípios que já vimos até aqui. E finalmente, vamos estudar um pouco de interfaces.

Criando um novo pacote para meios de transporte

No Delphi 2010, clique em File>New>Package. Para facilitar, salve o novo pacote no mesmo diretório de nossa aplicação criada até aqui. Usando o Project Manager, no editor do pacote, dê um clique de direita no item contains e adicione todas as units relativas aos meios de transporte, ou seja, uMeioTransporte.pas, uAviao.pas e uCarro.pas. Ainda no Project Manager, dê um clique de direita sobre o grupo de projetos e adicione o pacote pkPessoa. Remova deste último pacote as units que contêm as classes relativas à meios de transporte, deixando somente a unit que define a classe (componente) TPessoa. Clique de direita no grupo e dê um Build em ambos os pacotes, confirmando as dependências.

Como agora a nossa classe (componente) TPessoa referencia um TMeioTransporte que está em outra unit em outro pacote, precisamos trabalhar na seção Requires. Dê um clique de direita sobre este item no pacote pkPessoa, escolha Add Reference, clique em browse e localize o pkTransporte.dcp, que deve estar por padrão na pasta C:\Users\Public\Documents\RAD Studio\7.0\Dcp (no caso do Windows Vista e Delphi 2010). Isto também pode ser feito dando-se agora um clique de direita no nome do pacote e escolhendo Dependencies. No grupo, dê um clique de direita e escolha Build All para compilar novamente ambos os pacotes. Instale o pacote de Transportes. Seu grupo de programas e pacotes devem agora estar semelhantes ao mostrado na **Figura 1**.

Como você deve ter notado, a seção contains de um pacote indica quais units estão dentro do pacote e serão compiladas no BPL. A seção requires indica que outros pacotes são necessários para que seja feito com sucesso o projeto de compilação. Requires indica dependência e é muito importante que você tome cuidado com isso. Se você construir um grande framework com várias classes distribuídas em vários pacotes, uma mudança em uma única classe pode fazer com que todo o framework seja recompilado. Além disso, se um pacote depende de outro, isso também afeta o deploy. Em nosso exemplo, você não vai conseguir usar o pacote pkPessoa sem distribuir e instalar o pkTransporte, pois em nossa modelagem, toda pessoa deve ser capaz de operar um meio de transporte. Porém, o inverso não é verdadeiro, podemos distribuir meios de transporte sem pessoas, porque não há uma relação de dependência nesse sentido. É nesse ponto que vemos a vantagem de usar classes abstratas e interfaces para diminuir impacto na mudança de interfaces. Poderíamos dar um passo além, como faz a VCL (com DataSources e DataSets), colocando as classes abstratas em um pacote separado e as implementações em outro. Em nosso exemplo, para fins didáticos, não usamos essa abordagem. A esta altura, você já deve estar familiarizado com pacotes e fica aí uma bela sugestão de exercício.

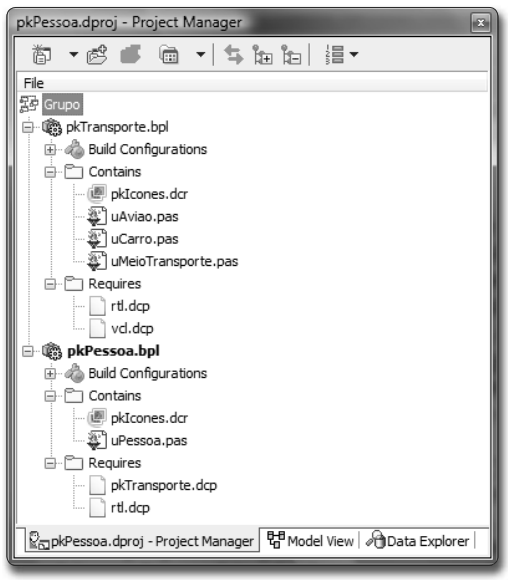


Figura 1. Pacotes pkPessoa e pkTransporte

Um outro cuidado com relação a dependências: o próprio IDE do Delphi faz uso desses BPLs, claro, porque eles são de design time e runtime (o padrão). Com isso, se você instalar pkPessoa, automaticamente o IDE vai instalar pkTransporte. Se retirar pkTransporte, pkPessoa não poderá ficar sozinho no IDE. Faça um teste com a VCL para comprovar, clique em Component>Install Packages e tente desinstalar o pacote Embarcadero Database Components, que contém classes core de acesso a dados da VCL como TDataSource, TDataSet etc. Como todas as implementações de acesso a dados no Delphi (BDE, ADO, DBX, ClientDataSet, controles Dara Aware) dependem dessas classes, todas serão retiradas da memória e não podem ser usadas. Veja o warning mostrado na **Figura 2 (**não confirme, clique em No, fizemos apenas um teste). Agora aproveitemos o exemplo para pensar na seguinte hipótese: e se por algum acaso a equipe do Delphi resolvesse modificar algo na interface de TDataSet, como incluir mais um método virtual? Todos os pacotes dependentes teriam que ser recompilados, incluindo de terceiros (que teriam que fornecer novas versões de seus pacotes). Ou seja, uma reação em cadeia devastadora. Lembro-

me até hoje de uma proeza que a equipe do Delphi fez, lançar o Service Pack 1 do Delphi 7 sem que para isso nenhuma interface fosse modificada, ou seja, nenhum pacote /unit dependente foi afetado. Por isso sempre digo, o que você escreve entre o Begin e o End não importa para efeitos de modelagem, o importante é acertar a interface! Jamais publique uma interface, uma classe abstrata, que possa estar suscetível a mudanças constantes. Existe uma regra na OO que diz: "Interfaces são imutáveis". Vale lembrar que aqui não estamos usando diretamente interfaces, mas sim classes abstratas (TMeioTransporte), mas seu papel e importância é equivalente.

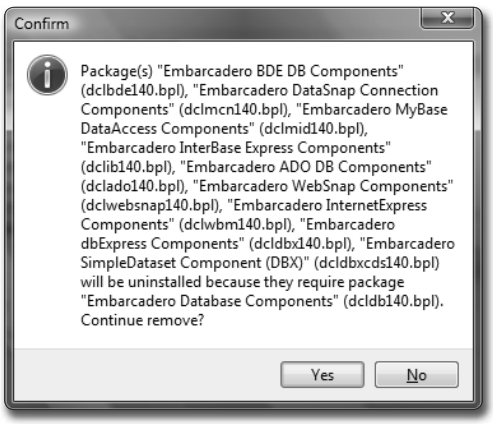


Figura 2. Tente remover o pacote base de acesso a dados do IDE para ver o que acontece

UML

Desde o Delphi 2005, o suporte à modelagem UML foi incluído no IDE. Com isso, podemos criar diagramas de classes e outros diagramas UML dentro do próprio ambiente. Para nosso exemplo, nada melhor do que visualizarmos como ficou nosso framework de transporte e pessoas em um diagrama, fazendo engenharia-reversa. Aliás, poderíamos ter feito tudo isso desde o começo, mas deixei para apresentar a UML justamente agora. Poderíamos ter criado um diagrama de classes novinho, visualmente criado as classes, seus atributos, métodos, e o Delphi geraria o respectivo código-fonte, inclusive com as relação de herança (Transporte<-Carro/Aviao) e associação (Transporte-Pessoa).

No editor do pacote, clique de direita em qualquer um dos pacotes e escolha Modeling Support. Marque os dois pacotes (**Figura 3**) e clique em Ok. Será criado um diretório especial na pasta do projeto/pacotes para incluir o suporte à modelagem. Agora clique em View>Model View, no editor que aparece clique de direita na parte branca e escolha Add>Class Diagram. Arraste as classes para o diagrama. Veja o resultado na **Figura 4**.

Estendendo o framework

Vamos neste ponto do artigo criar mais uma classe descendente de TMeioTransporte, para comprovar que nosso framework é expansível, assim como a VCL, reforçando a abstração, encapsulamento e polimorfismo.

Figura 3. Adicionando suporte à modelagem

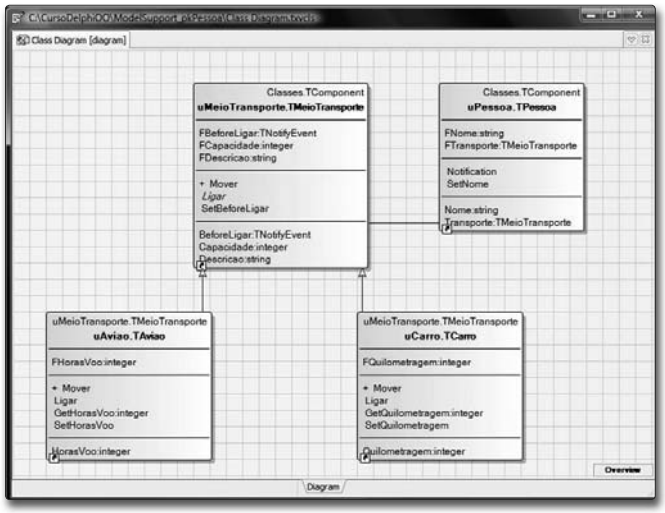


Figura 4. Diagrama de classes - framework de transportes

Com o pacote pkTransporte aberto, crie uma nova unit no Delphi e implemente-a conforme mostrado na **Listagem 1**. Note que aqui estamos criando uma classe descendente de TMeioTransporte (TBike), logo, devemos implementar seus métodos polimórficos. Além disso, adicionei uma propriedade

Nota do DevMan

A primeira implementação do MIDAS (Multitier Distributed Application Services), lançada originalmente no Delphi 4 (e um protótipo no Delphi 3), tinha um erro grave que feria o princípio que citei, que interface são imutáveis. Quando você colocava um DataSetProvider em um DataModule remoto, um novo método de interface (get e set) era criado para que fosse possível a comunicação entre cliente e servidor e troca de Data Packets. Ou seja, uma nova tabela no banco, novo DataSetProvider. E sabe o que ocorria quando você colocava um novo DataSetProvider? Um novo método era criado, logo, a interface COM era modificada, logo, aplicações clientes que acessam aquele servidor também precisariam ser recompiladas. IAppServer, no Delphi 5, resolveu tudo isso. Não são criados novos métodos para cada tabela, pois tudo pode ser trocado através de apenas 7 métodos básicos da interface, que trabalham com todos os tipos de providers. É por esse motivo que você coloca um DataSetProvider em um DataModule, seja ela remoto ou não, e automaticamente já o enxerga na propriedade ProviderName de um ClientDataSet.

extra à classe, chamada Suspensao (booleana), para indicar se a bicicleta tem esse acessório. Salve a unit como uBike.pas. Codificada a classe, basta recompilar tudo que o pacote será reinstalado e o novo componente vai aparecer no IDE.

Listagem 1. Nova classe: TBike

```
unit uBike;

interface

uses
  // coloque essa unit p/ acessar a classe TMeioTransporte
  uMeioTransporte, Classes;

type
  // observe que TBike herda de TMeioTransporte
  TBike = class(TMeioTransporte)
  private
    FSuspensao : boolean;
    function GetSuspensao: boolean;
    procedure SetSuspensao(const Value: boolean);
  protected
    procedure Ligar(); override;
  public
    procedure Mover(); override;
  published
     property Suspensao: boolean
     read GetSuspensao write SetSuspensao;
  end;

  procedure Register;

implementation

uses Dialogs;
{ TCarro }

procedure Register;
begin
  RegisterComponents('ClubeDelphi',[TBike]);
end;


function TBike.GetSuspensao: boolean;
begin
   result := FSuspensao;
end;

procedure TBike.Ligar();
begin
  // repare que não vai inherited aqui
  // pois não existe nada na classe base
  ShowMessage('Colocando o pé no pedal...');
end;

procedure TBike.Mover();
begin
  inherited;
  ShowMessage(Descricao + ' entrou em movimento.');
end;

procedure TBike.SetSuspensao(const Value: boolean);
begin
   FSuspensao := Value;
end;

end.
```

Aqui finalizamos nosso pequeno framework de transporte. Vamos agora examinar um último e importante recurso da OO, interfaces. Para isso, vamos fazer um novo e simples exemplo. Na próxima e última parte deste curso veremos como usar todos os conhecimentos adquiridos nestas cinco partes do curso para criar uma aplicação real usando programação orientada a objetos com o Delphi.

Interfaces

Para finalizar este etapa do nosso mini-curso, vamos fazer uma introdução ao uso de interfaces. Este assunto já foi bastante abordado aqui mesmo na ClubeDelphi, de forma que vou procurar dar uma visão simples desse recurso que é amplamente utilizado na orientação a objetos.

Eu poderia citar aqui a variedade enorme de formas na qual você poderia utilizar interfaces, entre algumas:

- Na programação de objetos distribuídos - server e cliente assinam um "contrato" e se comunicam através dele. Neste contrato estão definidos tipos, métodos e protocolos;
- Aplicações DataSnap com banco de dados (ou sem);
- Criação de plug-ins para o IDE do Delphi;
- Criar sistemas mais extensíveis e com arquitetura plugável (como o IDE do Delphi);
- Abstrair totalmente uma classe de uma implementação, por exemplo, declarando uma propriedade de um tipo interface e não uma classe base ou abstrata;
- Simular herança múltipla, pois uma classe pode implementar mais de uma interface e ainda delegar a implementação delas a outras classes internas;
- Usar interfaces com polimorfismo (uma mistura poderosíssima);
- Criar pequenos elos de ligação entre classes, para torná-las o menos dependente possíveis, e facilmente substituíveis;
- Comunicar aplicações diferentes usando métodos conhecidos, por exemplo, acessar funcionalidades em uma DLL sem usar rotinas básicas como "external". O mesmo pode ser feito com pacotes de runtime. Exemplos no Delphi são "Import Type Library" (no caso do COM) ou "Import WSDL" (Web Services), ou ainda "Add (Web) Reference" no Dephi Prism;
- Podem ser usados para implementar contagem de referência de objetos e evitar memory leaks.

Uma interface é semelhante a uma classe que possua somente métodos abstratos, ou seja, sem implementação. Uma interface apenas define métodos que depois devem ser implementados por uma classe. Dessa forma, um objeto pode se comunicar com o outro apenas conhecendo a sua interface, que funciona como uma espécie de contrato (veja a **Figura 5**).

Figura 5. Classes e Interfaces

Uma interface é como se fosse um controle remoto. Você consegue interagir com um objeto conhecendo o que ele oferece, tendo a interface que descreve cada função, porém, sem a mínima ideia de como ele implementa essa funcionalidade internamente. Assim como TObject é a classe base para todas as classes do Delphi, a interface base para todas as interfaces é IInterface**.** Veja na **Listagem 2** a declaração de IInterface.

Os métodos _AddRef e _Release definem o mecanismo de contagem de referência. Isso significa que você não precisa liberar um objeto que implementa IInterface. QueryInterface faz solicitações dinamicamente a um objeto para obter uma referência para as interfaces que ele suporta.

Listagem 2. IInterface – interface base do Delphi

```
IInterface = interface
  ['{00000000-0000-0000-C000-000000000046}']
  function QueryInterface(const IID: TGUID; out Obj):
   HResult; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
end;
```

Exemplo usando Interfaces

Para demonstrar o uso de interfaces vamos criar um pequeno exemplo. Inicie uma nova aplicação no Delphi. Salve o formulário como "uFrmMain.pas" e o projeto "Interfaces.dpr". Dê o nome de "FrmMain" ao formulário. Abra a unit do formulário e declare a interface da **Listagem 3** na seção type. Note que não podemos apertar Shift+Ctrl+C e implementar o método Multiplicar. Esse é um método de interface, como um método abstrato de uma classe. Logo abaixo da interface declare o código da **Listagem 4.**

Listagem 3. Declarando uma interface

```
type
  ICalc = interface
    // para gerar o GUID abaixo, aperte SHIFT-CTRL+G
    ['{23326572-461F-4571-925E-63DD15AA51D2}']
    function Multiplicar (const x,y : integer) : integer;
  end;
```

Listagem 4. TComputador sabe calcular, TCalculadora também

```
TComputador = class (TInterfacedObject,ICalc)
  function Multiplicar (const x,y : integer) : integer;
end;

TCalculadora = class (TInterfacedObject,ICalc)
  function Multiplicar (const x,y : integer) : integer;
end;
```

Nota

A declaração TComputador = class (TInterfacedObject,ICalc) significa "TComputador herda de TInterfacedObject e implementa a interface ICalc". Isso não é herança múltipla.

Observe que ambas as classes TComputador e TCalculadora implementam a interface ICalc e descendem de TInterfacedObject. TInterfacedObject se encarrega de implementar a interface IInterface, logo, não precisamos implementar os métodos que vimos na **Listagem 2**. Aperte Shift+Ctrl+C para declarar os cabeçalhos dos métodos e implemente-os conforme a **Listagem 5**. Observe que ambas as classes implementam o método Multiplicar de ICalc, porém de formas diferentes.

Boa Prática

Em um exemplo real, declare a interface em um arquivo (unit) à parte, as classes que a implementam em outra, e obviamente as classes que as consomem em outra.

Nota

Note que toda interface tem um GUID – Globally Unique Identifier, um ID único que identifica uma interface. É impossível gerar dois GUIDs idênticos no mundo.

Usando Edits, Labels, um RadioGroup e um Button construa o formulário mostrado na **Figura 6.** Dê um duplo clique no botão e digite o código mostrado na **Listagem 6**. Observe que interessante é o código. Declaramos um objeto que é de um tipo interface. Depois, o criamos a partir de uma implementação / classe concreta, que pode ser qualquer classe que implemente ICalc, neste caso TComputador e TCalculadora. Chamamos então o método Multiplicar, que como foi declarado na interface, permite ser chamado sem que type-casts sejam feitos (o que faria o uso de interfaces perder o sentido). Rode e teste a aplicação, veja o resultado na **Figura 7.**

Algumas coisas a mais que você precisa saber sobre a técnica aqui demonstrada:

• Reforçando, interfaces são imutáveis. Depois que você publica uma interface, ela não pode ser mais modificada. Não que isso não possa ser feito, mas provavelmente é a pior prática da OO. Imagine a equipe do Delphi modificar a interface

Listagem 5. Implementando métodos

```
function TComputador.Multiplicar(const x, y: integer): integer;
begin
  result:=x * y;
end;

function TCalculadora.Multiplicar(const x, y: integer): integer;
var
  i : integer;
begin
  result := 0;
  if (x <> 0) and (y <> 0) then
    for I := 1 to abs(y) do
      result := result + x;
end;
```

Listagem 6. OnClick do botão do formulário

```
procedure TFrmMain.Button1Click(Sender: TObject);
var
  Obj : ICalc;
  n1,n2,r : integer;
begin
  if RadioGroup1.ItemIndex = -1 then exit;
  case RadioGroup1.ItemIndex of
    0 : Obj := TCalculadora.create;
    1 : Obj := TComputador.create;
  end;
  n1 := StrToInt(Edit1.Text);
  n2 := StrToInt(Edit2.Text);
  r := Obj.Multiplicar(n1,n2);
  Label2.Caption := IntToStr(r);
end;
```

Nota do DevMan

Uma curiosidade. Um dos frameworks que mais faz uso efetivo de interfaces é o WebSnap (framework lançado no Delphi 6 para desenvolvimento Web, sucessor do WebBroker). Você pode encontrar seus fontes na VCL do Delphi. A propósito, a classe que mais implementa interfaces no Delphi é desse framework, observe sua declaração:

TCustomAdapter = class(TComponent, IIdentifyAdapter, IWebVariableName, IWebVariablesContainer, INotifyList, IWebSnapObjClass, IGetAdapterErrors, IGetAdapterErrorsList, INotifyWebActivate, IAdapterEditor, IGetAdapterFields, IGetAdapterActions, IIteratorSupport, IClearAdapterValues, IEchoAdapterFieldValues, IAdapterAccess, IGetAdapterHiddenFields, ICreateActionRequestContext, IWebDataFields, IWebActionsList, IAdapterNotifyAdapterChange, IIteratorIndex)

Figura 6. Uma aplicação que usa interfaces

Figura 7. Usando interfaces

IAppServer? Imagine alguém modificar o padrão USB para conexão de dispositivos?

• Poderíamos aqui neste exemplo usar Padrões de Projeto, como o Factory e o Abstract Factory, para delegar a criação da classe concreta a uma "fábrica" que cria classes concretas, tornando nossa arquitetura mais poderosa.

Conclusão

Aqui finalizamos nosso framework de meio de transportes. Comprovamos como ele é expansível, quando adicionamos mais uma classe ao framework. Examinamos como distribuir classes em pacotes, de forma organizada, e a importância da dependência entre eles. Finalmente, um overview simples sobre interfaces. Na próxima e última parte deste curso, vamos criar uma arquitetura real, uma aplicação Delphi que implementa e usa praticamente todos os recursos vistos até aqui. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Cursos Online

Web Mobile PLUS

A Revista **WebMobile** oferece para seus assinantes uma série de Cursos Online de alto padrão de qualidade .

Conheça abaixo os cursos já disponíveis:
www.devmedia.com.br/curso/webmobile

- Curso em .NET: **Aplicação completa de orçamento Doméstico no Visual Studio 2005**
- Curso em Java: **Introdução ao desenvolvimento para celulares com J2ME**

A sua melhor opção de aprendizagem!

Assine a **WebMobile** e
Comece já seu treinamento!
www.devmedia.com.br/assine

DevMedia GROUP Mais Informações: www.devmedia.com.br/central - Tel.: 21 3382-5038/ 3382-5025

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP

PDO – PHP Data Objects

Interface de acesso a Banco de Dados

Resumo do DevMan

PHP Data Objects (PDO) é uma interface de acesso a banco de dados. É uma extensão relativamente nova e tem como objetivo padronizar o acesso a banco de dados usa ndo as mesmas funções independentemente do banco que será realizado a conexão e manipulação de dados. A mesma está disponível desde a versão 5.1 do php, ou até mesmo em sua versão 5.0.

Rafael Silva
rafael@rlsystem.com.br
Trabalha com análise e desenvolvimento de sistemas. Atua na área de desenvolvimento desktop utilizando Delphi, C#, e foco na Web com PHP, ASP.NET, CSS, AJAX, sempre utilizando os padrões da Web 2.0 em seus projetos. Utiliza banco de dados MySQL, SQL Server, PostGreSQL e Firebird. Para outras informações acesse o site http://www.rlsystem.com.br.

Todos nós programadores PHP estamos acostumados a utilizar funções específicas para cada banco de dados que manipulamos em nossas aplicações Web. No banco de dados Mysql, por exemplo, temos disponíveis as funções mysql_connect, mysql_select_db, mysql_query, mysql_fetch_array, entre outras, assim como no

Do que trata o artigo
Apresentar a extensão PDO do PHP, utilizando alguns exemplos de manipulação com a mesma. Com a PDO temos uma interface genérica de acesso a banco de dados. Assim ela gera suas instruções SQL automaticamente, não havendo problemas para migrar de um banco de dados de sua aplicação para outro. Caso necessário, você pode se conectar a diferentes bancos de dados utilizando sua mesma estrutura lógica e de nomenclaturas.

Para que serve
Uma das grandes vantagens do PHP sempre foi a simplicidade para a manipulação de dados, mas até a sua versão 4 tínhamos um grande problema: existiam funções específicas para cada banco de dados. A PDO chegou para contornar esse problema.

Em que situação o tema é útil
Quase todos nós desenvolvedores/analistas já passamos por situações com clientes, empresas que precisaram migrar o banco de dados de sua aplicação. Isso é muito comum, não só em PHP, mas como qualquer outra linguagem. No artigo será apresentado a PDO, uma extensão que gera suporte para a maioria dos banco de dados do mercado. A mesma é nativa do PHP.

Postgresql, onde temos disponível a pg_connect, pg_select_db, pg_query, pg_fetch_array, e como em vários outros bancos de dados disponível na linguagem PHP. Imagine quando desenvolvemos uma grande aplicação para uma empresa, e por algum motivo de verbas, desempenho ou qualquer outra situação que aconteça, tenhamos que migrar para outro banco de dados. Com certeza seria um grande problema e praticamente todo sistema precisaria ser revisado e modificado, pois apesar das funções conterem nomenclaturas similares, algumas delas necessitam de informações diferentes em alguns de seus parâmetros. O objetivo do artigo é apresentar a extensão PDO do PHP e com isso alguns desses problemas serão com certeza contornados. Nos tópicos a seguir serão abordadas as principais características da PDO, instalação, conexões, exceções, transações etc.

Instalação e configuração

Antes de começarmos a utilizar a PDO, é necessário que a mesma esteja disponível em nosso servidor Web ou local. Hoje em dia por padrão ela já vem disponível em quase todos servidores de hospedagem que utilizam a versão 5.1.x do PHP. De qualquer forma é de extrema importância que você entre em contato com seu provedor e verifique se a extensão está instalada. Aqui iremos abordar a instalação da PDO em um servidor Windows, que utiliza um pacote de instalação denominado Wamp Server. É claro que não é uma exigência do mesmo, o requisito mínimo é você ter em sua máquina o Apache, PHP 5 e MySQL.

Nota

Caso seja de seu interesse realizar a instalação de um servidor local em sua máquina, recomendo o Wamp Server. Acesse o site (veja sessão Links) e realize o download do mesmo. Ele é um pacote de componentes que instala em sua máquina o Apache 2.2, PHP 5.2.8 e Mysql 5.1 em alguns cliques.

Todas as configurações de funcionamento do PHP - isso inclui principalmente extensões que ele deve carregar na sua inicialização - se encontram no arquivo de configurações PHP.ini. É neste arquivo que devemos indicar que o mesmo carregue a extensão PDO. Se você estiver utilizando o pacote de instalação Wamp Server, o arquivo de configuração php.ini se encontra no diretório C:\wamp\bin\apache\Apache2.2.11\bin\PHP.ini. Caso sua instalação tenha sido realizada manualmente, por padrão o arquivo se encontra no diretório C:\Windows\PHP.ini.

Abra o mesmo e procure a linha extension=php_pdo.dll, certamente você irá encontrar um ponto e vírgula(;) no início da linha, indicando que é um comentário. Retire o ";" caso haja (veja **Figura 1**). Com a linha descomentada você já pode salvar as alterações e reiniciar o Wamp Server (ou Apache). Somente essa configuração ainda não nos dá a possibilidade de utilizar a PDO. Além desta extensão, é necessário habilitar as extensões PDO para os bancos de dados que desejamos utilizar. Você pode observar que além da extensão php_pdo.dll, existem outras como, php_pdo_firebird.dll para Firebird, php_pdo_mssql.dll para Sql Server, php_pdo_mysql.dll para MySql, php_pgsql.dll para Postgresql etc. É claro que estas são algumas das extensões, você irá notar que existem outras extensões.

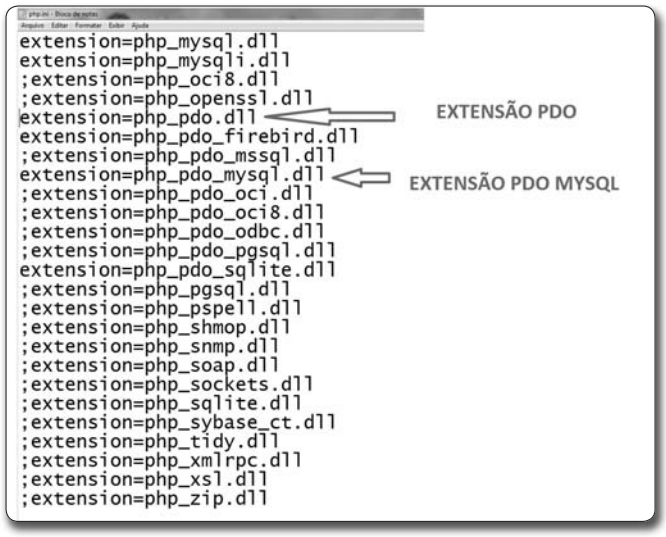


Figura 1. Arquivo PHP.ini

Como no artigo serão realizados exemplos com MySQL, Sql Server, Firebird e PostGreSQL, é necessário que as mesmas estejam habilitadas, seguindo a dica de instalação que foi mostrada anteriormente. É importante salientar que essas extensões devem estar fisicamente no diretório normalmente chamado de "ext" do php. Mas não se preocupe, no momento que você instala o Wamp Server, ou realizado o download do PHP direto do site oficial, todas elas já constam no diretório. De qualquer forma fica a dica caso ocorra um erro ao inicializar o PHP. Com essas pequenas configurações você já estará apto a utilizar a biblioteca PDO.

Conexões

Depois de comentados alguns conceitos sobre a PDO, vamos abordar na prática as conexões com bancos de dados. Como foi citado anteriormente, a PDO suporta os principais bancos de dados do mercado, nos exemplos a seguir serão realizadas conexões com os principais: Mysql, Sql Server, Firebird e Postgresql. Para iniciarmos é importante você saber que no construtor da classe PDO é necessário informar o DSN (Data Source Name). Na **Listagem 1** você confere a sintaxe do construtor da classe PDO.

Veja que no primeiro parâmetro é informada a string de DNS (Data Source Name), seguido do usuário do banco de dados, senha e o último que é opcional, um array com especificações de opções da conexão. Você poderá obter mais informações sobre o construtor da classe PDO na seção Links no final do artigo. Na **Listagem 2** você confere exemplos de conexão, já com os parâmetros necessários para os bancos citados.

Note como foi simples realizar conexões com PDO. Todos são muito similares, mudando apenas o nome do Driver do banco que será realizado a conexão.

Listagem 1. Construtor da classe PDO

```
# Sintaxe construtor da classe PDO #.
PDO::__construct ( string $dsn [, string $username [, string
$password [, array $driver_options ]]] )
```

Listagem 2. Exemplos de conexão

```
# Conexão MySQL #
$db = new PDO('mysql:host=localhost;dbname=cadastro', 'root', '123');

# Conexão Firebird
$db = new PDO ("firebird:dbname=localhost:C:/test/database/test.
FDB", 'SYSDBA', 'masterkey');

# Conexão SQL Server
$db = new PDO ('mssql:host=localhost;dbname=test', "pc", "123");

# Conexão PostGreSQL
$db = new PDO ('pgsql:host=localhost;dbname=postgres', "pg", "123");
```

Nota do DevMan

DSN - Data Source Name - é um registro que armazena as informações necessárias para fazer a conexão com o banco de dados, normalmente usado para guardar informações de conexão da tecnologia ODBC (Open Data Base Connectivity) tais como o tipo de banco de dados, driver, nome do banco de dados, diretório, o usuário, senha e outros dados usados para conexão.

Exceptions

Tratar exceções em uma aplicação tanto Web como Desktop é muito importante. Antes, nas funções utilizadas para realizar conexão como mysql_connect, não estavam disponíveis try e catch. Normalmente utilizávamos um or die para surgir uma mensagem quando ocorria um eventual erro, tanto em uma conexão ou uma execução de uma query. Na **Listagem 3** você visualiza um exemplo de exceção que é disparada caso aconteça um erro ao tentar conectar ao banco de dados MySQL.

Note que o catch recebe como parâmetro a classe PDOException. Utilizamos o método getMessage() para exibir o erro que a exceção nos retornou, caso haja. Na **Listagem 4** você encontra todas as propriedades e métodos da classe PDOException.

Prepared statements

A segurança na Web sempre foi uma grande preocupação para nós desenvolvedores, principalmente quando se tratava de sistemas de login, ou informações que são recebidas diretamente por usuários de nossos sites, normalmente por query string ou formulários. É muito comum encontrarmos sites com grandes vulnerabilidades, sem segurança alguma, onde qualquer usuário mal intencionado consegue realizar operações que podem ser prejudiciais ao site e até mesmo ao usuário final que esteja acessando.

No PHP, por exemplo, existem muitas aplicações Web com esses problemas, é comum encontrarmos sites que exijam login e com alguns comandos SQL Injection o mesmo já nos dá permissão de uso. Programadores PHP, por exemplo, normalmente utilizam as famosas funções addslashes, htmlentities, stripslashes e outros tipos de funções que eram criadas em conjunto com a str_replace. Com a classe PDO, já temos possibilidade de utilizar parâmetros em nossas query's. Na Mysqli, por exemplo, outra extensão de acesso a dados específica para Mysql, também é possível usar recurso semelhante (na edição 108 há um artigo que mostra uma aplicação com DAO e Mysqli para manipulação de informações no banco de dados.).

Trabalhar com parâmetros em nossas strings SQL é com certeza a melhor forma de nos prevenirmos. Você precisa ter muito cuidado com os dados recebidos, para isso é necessário utilizar algumas funções de escape de caracteres, como escape de aspas simples, aspas duplas, barras para garantir que nenhum usuário malicioso consiga injetar código em suas query's. Quando se está utilizando prepared statments isso não é necessário. A separação dos dados permite que se leve em conta automaticamente esses caracteres e eles não precisam ser tratados por nenhuma função especial.

Prepared statements podem ser utilizados para uma infinidade de razões, mas não devem ser utilizados em toda sua

Listagem 3. Tratando erros de conexão

```
<?php
try {
    $dbh = new PDO('mysql:host=localhost;dbname=cadastrox',
     'root', '123');
} catch (PDOException $e) {
    print "O Sistema encontrou um erro, erro original: " .
      $e->getMessage() . "<br/>";
}
?>
```

Listagem 4. Propriedades e métodos da classe PDOException

```
PDOException extends RuntimeException {

# Propriedades #

public array $errorInfo ;
protected string $message ;
protected string $code ;

# Métodos #
final public string Exception::getMessage ( void )
final public int Exception::getCode ( void )
final public string Exception::getFile ( void )
final public string Exception::getLine ( void )
final public array Exception::getTrace ( void )
final public string Exception::getTraceAsString ( void )
public string Exception::__toString ( void )
final private string Exception::__clone ( void )
}
```

Nota do DevMan

O tratamento de exceção, na ciência da computação, é o mecanismo responsável pelo tratamento da ocorrência de condições que alteram o fluxo normal da execução de programas de computadores. Em geral, na ocorrência de uma exceção, o estado do programa é gravado em um local pré-definido e a sua execução é direcionada para uma rotina de tratamento. Dependendo da situação, a rotina de tratamento pode prosseguir a execução a partir do ponto que originou a exceção, utilizando a informação gravada para restaurar o estado. Um exemplo de exceção que normalmente permite o prosseguimento da execução é aquela originada de uma falta de página (page fault). Por outro lado, uma exceção originada de uma divisão por zero usualmente impede o prosseguimento da execução do programa. Do ponto de vista do processador, as interrupções de hardware podem ser consideradas como condições análogas às exceções que permitem prosseguimento da execução, apesar destas exceções não serem normalmente relacionadas com o fluxo normal do programa.

aplicação. Algumas vezes podem ser mais lentos que uma consulta normal. A razão é que existem dois round-trips (”idas”) para o servidor, o que pode comprometer a performance para consultas que são executadas apenas uma vez.

Cabe ao desenvolvedor decidir se compensará ter impacto no desempenho para ganhar mais segurança. Depois de descrito alguns conceitos importantes, vamos construir um exemplo de comandos preparados. A ideia do exemplo é simples, vamos criar uma tabela denominada dados e a mesma vai conter apenas quatro campos, no caso id, nome, email e cidade. Na **Listagem 5** você confere o script SQL de criação da tabela. Lembrando que o script será executado no banco de dados “artigo”, você poderá criar qualquer um de sua preferência, o exemplo que será criado é meramente didático.

O objetivo principal é criar um código básico com os campos criados no banco de dados, que conterá a conexão e as rotinas de inserção. É claro que o ideal é você criar uma classe DAO para manipular essas informações de acesso a dados, conforme realizamos um artigo na edição 108 da ClubeDelphi, mas por ser um exemplo didático como citei, não vamos nos preocupar com essa prática. Veja na **Listagem 6** o exemplo construído.

Listagem 5. Script SQL da tabela dados

```
CREATE TABLE IF NOT EXISTS `dados` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`nome` varchar(50) NOT NULL,
`email` varchar(50) NOT NULL,
`cidade` varchar(30) NOT NULL,
PRIMARY KEY (`id`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
```

Listagem 6. Script SQL da tabela dados

```
<?
$db = new PDO('mysql:host=localhost;dbname=artigo', 'root',
'123');

$sql = $db->prepare("INSERT INTO dados (nome, email, cidade)
VALUES (:nome, :email, :cidade)");

$parametros = array('nome'=>'Rafael', 'email'=>'@@',
'cidade'=>'city');

$sql->execute($parametros);
?>
```

Note que primeiramente instanciamos a classe PDO e já no seu construtor informo a minha DNS (Data Source Name). Em seguida atribuímos à variável $sql nosso objeto $db junto com o método prepare, que irá preparar nossa SQL. Com certeza você deve ter notado uma diferença de nossa query para qualquer outra string sql que você já tenha visto em sistemas desenvolvidos em php, e até nossos últimos artigos da ClubeDelphi.

Assim como citado no início do tópico, estamos passando parâmetros para nossa query, com isso eliminamos a concatenação de strings, o que já deixa nossa aplicação segura (como citado anteriormente). Logo após criamos uma variável $parametros, que na verdade é um array, onde indicamos os devidos parâmetros da nossa query, junto com os valores, e no final é chamado o método execute, que irá executar nossa SQL, recebendo a variável $parametro que contém esses nossos valores a serem inseridos no banco de dados.

Nota do DevMan

Prepared statment serve para setar uma consulta e executá-la várias vezes com parâmetros diferentes. São desenvolvidos para queries ad hoc, são mais seguros e mais eficientes. Segue um exemplo simples:

SELECT * FROM email WHERE email = ?

O ? (ponto de interrogação) é chamado de placeholder. Quando a consulta acima for executada, é necessário passar um valor, que substituirá o ? na consulta.

Nota do DevMan

DAO (Data Access Object)

Atualmente é o modelo preferido para persistir dados em aplicações de banco de dados relacional. DAO (Data Access Object) é um padrão para persistência de dados que permite separar regras de negócio das regras de acesso a banco de dados. Numa aplicação que utilize a arquitetura MVC, todas as funcionalidades de bancos de dados, tais como obter as conexões, mapear objetos para tipos de dados SQL ou executar comandos SQL, devem ser feitas por classes DAO.

Transações

Talvez alguns desenvolvedores PHP ainda não tenham utilizado o recurso de transações, mas essa é sem dúvida uma das maiores inovações que considero da biblioteca PDO. Para ser bem sincero até era possível utilizar transações com as funções mysql_(...), mas de uma forma não muito convencional, onde na verdade utilizamos as função a mysql_query(---), como por exemplo mysql_query(“SET AUTOCOMMIT=0;”), mysql_query(“START TRANSACTION;”), e depois chamávamos nossas instruções SQL por exemplo, e no final um commit na transação com mysql_query(“COMMIT;”).

Veja que essa era uma das formas que poderíamos utilizar transações no PHP, uma forma nada agradável é verdade. Mas agora com a PDO já temos uma maneira simples e profissional de utilizar transação. Vamos imaginar o seguinte cenário: temos uma pequena tabela de clientes, onde ela possui alguns campos, entre eles o campo saldo, e precisamos transferir R$100,00 do João para a conta da Maria, por exemplo. Basicamente o que iríamos fazer? Primeiramente realizar um update em Maria, acrescentando R$100,00 em seu saldo, logo após outro update debitando R$100,00 de João. E se por ventura, tenhamos um problema e ele somente atualize o saldo da Maria, e deixa o João com o mesmo valor, não havendo nenhum débito do mesmo? Simplesmente estaria gerando um grande erro no nosso sistema, já que a transação bancária não estaria sendo realizada com sucesso, o que acabaria dando desfalque na empresa. É claro que isso é um exemplo muito simples, mas que de forma objetiva mostra quando e porque devemos realizar transações com banco de dados.

Na **Listagem 7** temos um exemplo de uma tabela de clientes. Na **Listagem 8** você visualiza o exemplo construído para esse nosso cenário apresentado. O código é muito simples,

primeiramente realizamos a conexão com o banco de dados, em seguida é chamado o método beginTransaction, que irá neste caso iniciar uma transação com o MySQL. Em seguida, utilizando o método exec, passamos como parâmetros nossas strings SQL, a primeira colocando um saldo de R$ 100,00 na cliente Maria, e a segunda debitando R$ 100,00 do cliente João.

Listagem 7. Script SQL da tabela clientes

```
CREATE TABLE IF NOT EXISTS `clientes` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `nome` varchar(30) NOT NULL,
  `saldo` decimal(10,2) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=3 ;

INSERT INTO `clientes` (`id`, `nome`, `saldo`) VALUES
(1, 'João', '200.00'),
(2, 'Maria', '50.00');
```

Listagem 8. Exemplo de transação com a PDO

```
<?
try
{
  # Istancia PDO #
  $pdo = new PDO('mysql:host=localhost;dbname=artigo', 'root',
   '123');

  # Inicio a transação #
  $pdo->beginTransaction();

  # SQL Update no cliente Maria#
  $pdo->exec("UPDATE clientes SET saldo = saldo + 100 WHERE id = 2");

  # SQL Update no cliente João#
  $pdo->exec("UPDATE clientes SET saldo = saldo - 100 WHERE id = 1");

  # Commita #
  $pdo->commit();

} catch (Exception $e)
{
  $pdo->rollBack();
  echo "Erro: " . $e->getMessage();
}
?>
```

Veja que, se ocorrer algum erro, o mesmo vai para o catch, onde é chamado o método rollBack, que desfaz a transação. Caso não aconteça nenhum erro, a transação é comitada e a operação é realizada com sucesso e sem perigo algum de eventuais erros. O exemplo aqui é muito simples, mas será sua base para possíveis implantações no seu sistema. Note a facilidade de se trabalhar com transações com PDO. Detalhe: isso vale para qualquer banco de dados que a PDO dá suporte, não sendo necessária nenhuma alteração sequer em nossos sistemas numa possível migração, como citado no começo do artigo.

Conclusão

A extensão PDO ainda é pouco utilizada, infelizmente. Isso com certeza se deve ao fato da maioria dos programadores ainda não conhecerem bem a orientação a objetos. No entanto, dominando o que apresentamos neste artigo, você será capaz de criar grandes sistemas, robustos e com grande performance, e sem contar a facilidade de migração de banco de dados que você tem. ●

Nota do DevMan

Uma transação é um conjunto de procedimentos que é executado num banco de dados, que para o usuário é visto como uma única ação. A integridade de uma transação depende de quatro propriedades, conhecidas como ACID.

Atomicidade - Todas as ações que compõem a unidade de trabalho da transação devem ser concluídas com sucesso, para que seja efetivada. Qualquer ação que constituir falha na unidade de trabalho deve ser desfeita (rollback). Quando todas as ações são efetuadas com sucesso, a transação pode ser efetivada (commit).

Consistência - Nenhuma operação do banco de dados de uma transação pode ser parcial. O status de uma transação deve ser implementado na íntegra. Por exemplo, um pagamento de conta não pode ser efetivado se o processo que debita o valor da conta corrente do usuário não for efetivado antes, nem vice-versa.

Isolamento - Cada transação funciona completamente à parte de outras estações. Todas as operações são parte de uma transação única. O princípio é que nenhuma outra transação, operando no mesmo sistema, pode interferir no funcionamento da transação corrente (é um mecanismo de controle). Outras transações não podem visualizar os resultados parciais das operações de uma transação em andamento.

Durabilidade - Significa que os resultados de uma transação são permanentes e podem ser desfeitos somente por uma transação subsequente. Por exemplo: todos os dados e status relativos a uma transação devem ser armazenados num repositório permanente, não sendo passíveis de falha por uma falha de hardware.

Links

WAMP Server – Servidor com Apache, PHP 5 e MySQL 5
http://www.wampserver.com

Manual PDO
http://br.php.net/PDO

Stored Procedures com a PDO
http://br.php.net/manual/pt_BR/pdo.prepared-statements.php

PDO Exception
http://www.php.net/manual/en/class.pdoexception.php

PDO Drivers
http://www.php.net/manual/en/pdo.drivers.php

PDO Constructor
http://www.php.net/manual/pt_BR/pdo.construct.php

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Existem momentos onde a estrutura faz toda a diferença.

Se o tempo é ruim, você encontra na Alog um porto seguro. Isto porque sua empresa reduz custos e melhora a governança em TI. Entre em contato e descubra porque a Alog é líder em hosting gerenciado.

REDUÇÃO DOS CUSTOS AUMENTO DA GOVERNANÇA EM TI

0800 282 3330 | alog.com.br

ALOG
DATA CENTERS DO BRASIL

HOSTING GERENCIADO | COLOCATION | CONTINGÊNCIA | E-MAIL CORPORATIVO | CONECTIVIDADE | SERVIÇOS PROFISSIONAIS

Hospede seus sites de forma simples, rápida e ilimitada.

Plano **Básico**

- 1 domínio principal
- 50 domínios hospedados
- Transferência **ilimitada**
- Espaço **ilimitado**
- 10 contas de e-mail
- PHP, ASP e ASP.NET
- Painel de controle Plesk
- Suporte 24hs

Tudo isso por apenas

R$ **15,90** por mês
pagamento trimestral

Assine agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

- Conheça também nossos planos profissionais com registro de domínio grátis e isenção de taxas anuais enquanto for cliente.

HOSPEDIX
HOSPEDAGEM PROFISSIONAL
www.hospedix.com.br

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe 30 dias inteiramente grátis.

RVSTMEDIA

As promoções anunciadas não são cumulativas com outras promoções e são válidas apenas para clientes Hospedix ativos.